2008-09

An approach for developing and validating libraries of temporal formal specifications

Sybor, Colleen A.

Monterey, California. Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

AN APPROACH FOR DEVELOPING AND VALIDATING LIBRARIES OF TEMPORAL FORMAL SPECIFICATIONS

by

James J. Sordi Jr.
Colleen A. Sybor

September 2008

Thesis Advisor:	James B. Michael
Co-Advisors:	Doron Drusinsky
	Man-Tak Shing

Approved for public release; distribution unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** September 2008	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE An Approach for Developing and Validating Libraries of Temporal Formal Specifications		**5. FUNDING NUMBERS**	
6. AUTHOR(S) James J. Sordi, Colleen A. Sybor			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000		**8. PERFORMING ORGANIZATION REPORT NUMBER**	
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A		**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**	
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; Distribution unlimited		**12b. DISTRIBUTION CODE**	

13. ABSTRACT (maximum 200 words)

This thesis examines the role of independent validation in the development of software systems. As software systems become increasingly larger and more complex the role of software validation becomes crucial. In particular, one must make sure that the specification of a software system is correct with respect to customer expectations. We introduce an approach for developing and validating reuse libraries of temporal formal specifications. These libraries include UML statechart based assertions for formal specifications and their associated validation test scenarios. We build the validation test scenarios with the goal of ensuring that specifications within the libraries are indeed error-free and consistent.

14. SUBJECT TERMS Validation, Reuse, Requirements, System Reference Model, Formal Specification, Assertions			**15. NUMBER OF PAGES** 101
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

AN APPROACH FOR DEVELOPING AND VALIDATING LIBRARIES OF TEMPORAL FORMAL SPECIFICATIONS

James J. Sordi Jr.
Lieutenant, United States Navy

Colleen A. Sybor
Lieutenant, United States Navy

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 2008

Authors: James J. Sordi Jr.
Colleen A. Sybor

Approved by:

James B. Michael
Co-Advisor

Doron Drusinsky
Co-Advisor

Man-Tak Shing
Co-Advisor

Peter J. Denning
Chairman, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

This thesis examines the role of independent validation in the development of software systems. As software systems become increasingly larger and more complex the role of software validation becomes crucial. In particular, one must make sure that the specification of a software system is correct with respect to customer expectations. We introduce an approach for developing and validating reuse libraries of temporal formal specifications. These libraries include UML statechart based assertions for formal specifications and their associated validation test scenarios. We build the validation test scenarios with the goal of ensuring that specifications within the libraries are indeed error-free and consistent.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

Micah thank you for all your help and support, we couldn’t have done it without you.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. MOTIVATION

Software is essential to almost every facet of our daily lives from business to science, and while the advantages are numerous and have arguably bettered our lives, it comes with a cost. The National Institute of Standard and Technology (NIST) sponsored a study in 2001 and found that the annual cost of software errors to the U.S. Economy in 2001 was approximately $59.5 billion.[1] Additionally the study found that over half the costs have been borne by the users. This is remarkable because software practitioners have not yet been held accountable to the same standards imposed on engineers in traditional engineering disciplines.

Over the years, some software defects have resulted in human injuries, property damage, and in extreme cases, loss of human lives. This is a cost that is unacceptable to users and must not be accepted by software developers. One of the most well known examples of software error causing human fatalities is the THERAC 25[2]. This machine was supposed to save human lives by sending the proper amount of radiation into patients, but instead it overdosed humans with massive

[1] National Institute of Standards and Technology (NIST), "Software errors cost U.S. economy $59.5 billion annually." National Institute of Standards and Technology (NIST2002-10) (2002), http://www.nist.gov/public_affairs/releases/n02-10.htm (accessed May 10, 2008).

[2] N. Levenson and C. Turner, "Investigation of the Therac 25 accidents." *IEEE Computer* (July 1993), 18-41.

amounts of radiation and resulted in several fatalities.[3] These failures are not just unacceptable but could have been avoided had the software been validated and verified properly. The IEEE standards, for validation and verification, help to guide the software developers with two main questions, “am I building the right product?” and “am I building the product right?” These questions are longstanding and will, if answered appropriately, help reduce the risk of mishaps due to software defects.

Another challenge the software industry faces is that software is increasing in complexity, making it difficult to detect errors and eliminate them. This also increases the importance of validation and testing methods that enable earlier and more effective error identification and removal. Software must be verified and validated to ensure not just quality and safety but also guard against waste, in terms of money and lives.

Our motivation for the research reported here is to develop techniques that improve the engineer’s ability to validate software systems. Additionally, these validation techniques will be applicable to the entire software industry and when used in conjunction with verification will hopefully result in better software systems and reduce software defects and their attendant costs.

[3] R. Merritt, “Embedded experts: fix code bugs or cost lives.” *Information Week* (April 10, 2006), http://www.informationweek.com/news/management/showArticle.jhtml?articleID=185300011 (accessed May 05, 2008).

B. INDEPENDENT VALIDATION AND VERIFICATION

The current guideline for validation and verification is the IEEE standard 1012-20044 which has had to evolve because of decades of unsuccessful software. The overall aim is to establish guidelines for the software industry to follow and help to create better software products.

The following is directly quoted from the IEEE standard 1012-2004[5]:

Software V&V processes consists of the following:

- Verification process and validation process. The verification process provides objective evidence whether the software and its associated products and processes conform to requirements (e.g., for correctness, completeness, consistency, accuracy) for all life cycle activities during each life cycle process acquisition, supply, development, operation, and maintenance)satisfy standards, practices, and conventions during life cycle processes.
- Successfully complete each life cycle activity and satisfy all the criteria for initiating succeeding life cycle activities (e.g., building the software correctly).
- The validation process provides evidence whether the software and its associated products and processes satisfy system requirements allocated to software at the end of each life cycle activity.
- Solve the right problem (e.g., correctly model physical laws, implement business rules, use the proper system assumptions).
- Satisfy intended use and user needs.

---

4 Institute of Electrical and Electronics Engineers, Standard for Software Verification and Validation,IEEE-STD-1012, June 08, 2005.

5 Institute of Electrical and Electronics Engineers, Standard for Software Verification and Validation,IEEE-STD-1012, June 08, 2005.

The verification process and the validation process are interrelated and complementary processes that use each other's process results to establish better completion criteria and analysis, evaluation, review, inspection, assessment, and test V&V tasks for each software life cycle activity.

The IEEE guidelines leave the software industry to develop their own validation and verification methods. The industry has yet to integrate these techniques fully, contributing to the poor record of software acquisition. In fact, the Standish Group reported that the success rate of software projects was 35% in 2006, and the report claimed that software developers fielded only 46% of the required features and functions, which means that the projects did not meet the needs of the customer.[6] These studies indicate conformance to the V&V guidelines are not enough to significantly lower the defect rate in software partition of systems. The V&V guidelines empower the individual to create and implement their own IV&V plan of actions, and the only consequence to not following the guidelines is unsuccessful acquisition of software. Formal V&V (FV&V) techniques can be used to address some of the failings of the existing practice of V&V.

C. THE NEED FOR A STANDARD TECHNIQUE

The current problem facing the software industry in facilitating validation is that there are no concise, simple techniques to conduct validation. The IEEE IV&V guidelines are general and meant to guide the industry in the actual

[6] The Standish Group International, "Annual Chaos Report." (2006).

implementation of validation. This leaves the software industry to its own devices on the implementation of validation and can result, in the worst case scenario, with poor validation processes if validation is conducted at all. This is largely due to the ignorance of validation procedures and misunderstanding of the necessity of validation. And it has resulted in the industry's primary focus on verification because it can be easier to accomplish and minimal effort is put into validation. Thus much effort is in "have we built the system right?" but not "have we built the right system?"

Adding to the problem, there is no general consensus among the academic community on how to complete the validation phase, many different paths are used. The typical way for a system to be built, if there is structure at all, is for the software requirements to be gathered, using pen and paper, use cases built and then code is written directly from the use cases. There is no formal link from the requirements to the formal specification of the system behaviors (if one exists) to ensure that the correct system is being built. The formal specification of system behaviors includes assertions which precisely model the required behavior of the system and can be traceable to the system requirements providing a means to ensure that the correct system is being built. As examples, both Voyager and Galileo had significant software errors; the primary cause of the faults were directly related to system behaviors that had not been identified or developed by the developers. One can significantly reduce these kids of errors by formally

specifying the required behaviors in terms of assertions and validating the correctness of these assertions against stakeholder expectations before building the software.

One way to facilitate the validation process is through execution-based validation. Execution-based validation is the process of inferring certain behavioral properties to exercise the system under test (SUT) in a known environment and with selected inputs. This gives the person conducting validation the capability to validate that the system being built is the correct system based on user requirements. In our thesis we use the StateRover white-box automatic test-generator. The white-box test generator constructs a JUnit TestCase class from a given statechart assertion model and the associate embedded assertions. The advantages of this process include: the ability to pinpoint specific errors; investigate the causes of failures on a specific input in detail; and eliminate errors in their design in an efficient manner.

D. THE ROLE OF SOFTWARE REUSE

Software reuse is an important concept that can help clarify validation techniques, making them more relevant for software development teams. Software reuse aims to increase the productivity, efficiency and quality of software by reusing the applicable software from one project in another project.[7] By reusing the software the developers can save resources that would have otherwise been used to develop the software.

[7] W. Lim, *Managing software reuse, a comprehensive guide to strategically reengineering the organization for reusable components.* Upper Saddle River, New Jersey: Prentice Hall PTR, (1998): 7.

An important part of validation is the creation of formal specifications in the form of assertion statements to capture the correct behavior of the software from natural language requirements. Assertion statements can be difficult to define and produce because of natural language ambiguities. However, with the use of the reuse concepts, libraries can be established. These libraries will contain correct assertion statements which have been thoroughly tested. The assertion development team can then reuse the correct assertion statement and use the accompanying test suite to ensure that the chosen assertion matches the requirements and proper validation is occurring.

E. OUTLINE

This work's main objective is to facilitate the assertion validation process. This is accomplished through the use of libraries which contain consistent and accurate assertions. We intend to demonstrate that these assertions are correct and reusable through the use of testing scenarios. These assertions will provide a type of engineering control for the IV&V process.

The organization of the thesis is as follows:

Chapter II provides background information about IV&V and software reuse. The chapter will show a deficiency in the current guidance provided for software validation, and will describe reuse techniques that have been accomplished to date.

Chapter III discusses the NASA System Reference Model (SRM) and the use of assertions in repository libraries.

Chapter IV discusses the use of patterns to facilitate Software reuse.

Chapter V provides conclusions and recommendations for future research.

II. IV&V AND SOFTWARE REUSE

A. COMPLEXITY OF SOFTWARE DESIGN

Anyone who is associated with software design understands that software systems can be extremely complex. They are so complex that most software engineering researchers often focus their research solely on ways to deal with complexity. The reason these systems are so complicated can often be traced back to the users' requirements for that system. A system that is required to perform several functions will naturally be more complex than a system that is required to perform just one function. Common problems that occur when developing software include failing to match the final product to the customers' needs, or dealing with errors in the software that often reveal themselves at the worst possible time and are often costly to fix. One way to try to avoid these problems is to implement Independent Verification and Validation (IV&V) and software reuse into the development of the systems. This chapter covers IV&V, how IV&V is conducted, and how software engineers are currently leveraging software reuse in building software systems.

B. DEFINITIONS

Independent: Independence in relation to IV&V is defined by the Institute of Electrical and Electronics Engineers (IEEE) using three parameters: technical independence, managerial independence, and financial independence.

Technical Independence: "requires the V&V effort to utilize personnel who are not involved in the development of the software."[8]

Managerial Independence: "requires that the responsibility for the IV&V effort be vested in an organization separate from the development and program management organizations."[9]

Financial Independence: "requires that control of the IV&V budget be vested in an organization independent of the development organization."[10]

Verification: "The process of evaluating a system or component to determine whether the products of a given deployment phase satisfy the conditions imposed at the start of that phase." [11] Software verification answers the question, "Are we building the product right?"

Validation: "The process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements." [12]Software validation answers the question "Are we building the right product?"

---

[8] Institute of Electrical and Electronics Engineers, Standard for Software Verification and Validation,IEEE-STD-1012, June 08, 2005.

[9] Ibid.

[10] Ibid.

[11] Ibid.

[12] Ibid.

Software IV&V: “a series of technical and management activities performed by someone other than the developer of a system to improve the quality and reliability of that system and to assure that the delivered product satisfies the user’s operational needs.”[13]

Software Reuse: “the use of existing software artifacts in the development of other software artifacts with the goal of improving productivity and quality, among other factors.”[14]

Requirement Specification: “an organization's understanding (in writing) of a customer or potential client's system requirements and dependencies *at a particular point in time* (usually) prior to any actual design or development work.”[15]

Pattern: “a body of literature to help software developers resolve recurring problems encountered throughout all of software development.”[16]

C. IV&V BACKGROUND

In the early 1940s the first computer was developed to calculate artillery firing tables for the United States

---

[13] R. Lewis, *Independent verification & validation: A life cycle engineering process for quality software*. New York: John Wiley & Sons, (1992): xxiii.

[14] W. Lim, *Managing software reuse, a comprehensive guide to strategically reengineering the organization for reusable components*. Upper Saddle River, New Jersey: Prentice Hall PTR, (1998): 7.

[15] D. Le Vie, “Writing software requirements specifications” TECHWR-L (MAR 2007) http://www.techwrl.com/techwhirl/magazine/writing/softwarerequirementspecs.htm (accessed July 15, 2008).

[16] B. Appleton, "Patterns and software: essential concepts and terminology” CM Crossroads (2000), http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html (accessed September 20, 2008).

Army's Ballistic Research Laboratory. The design of the computer was focused primarily on hardware, not paying much attention to software. In fact, some would say in the early stages of computing, software was often ignored. The intention of computers at this time was to perform a single task. When the task was identified the computers were hard-wired to accomplish that task. With the role of software being so small the need for IV&V had not yet been recognized. However, as time passed and the role and cost of software grew, the need for IV&V became evident.

In the mid 1940s John Von Neumann came up with two concepts that would have a direct impact on software design. The first was known as “shared program technique.” “This technique states that the actual computer hardware should be simple and not need to be hand-wired for each program. Rather, complex instructions should be used to control the simple hardware, allowing it to be reprogrammed much faster.”[17]

The second concept he developed was called “conditional control transfer.” “This idea gave rise to the notion of subroutines, or small blocks of code that could be jumped to in any order, instead of a single set of chronologically ordered steps for the computer to read. The second part of the idea stated that computer code should be able to branch out based on logical statements such as “IF” (expression) “THEN,” and looped with others such as a “FOR”

17 C. Robat, “Introduction to software history.” The History of Computing Project (October 17, 2006), http://www.thocp.net /software/software_reference/introduction_to_software_history.htm (accessed June 11, 2008).

statement."[18] The use of these concepts, and others like them, allowed software to grow into a more significant part of computer design.

As software grew, so did the cost associated with it. In the 1950s, software's cost was only 20% of the overall system cost. In the 1980's, software costs rose to 80%. Today, software costs can be up to 95% of the overall system cost.[19] These rising costs forced software developers to find a way to save money.

In the late 1950s, one of the leading software developers was the Department of Defense (DoD). The DoD began to notice projects were consistently behind schedule, over budget, and did not provide the required performance. This was unacceptable not only for financial reasons but because software errors can lead to loss of life, injury, or loss of property especially in military systems. The DoD was repeatedly surprised by the costly projects because "...software development contractors often gave overly optimistic assessments of the software development status to the DoD."[20] To address this, the DoD launched a plan to conduct IV&V on their software systems in an attempt to get accurate assessments of how their projects were doing. The

---

[18] C. Robat, "Introduction to software history." The History of Computing Project (October 17, 2006), http://www.thocp.net /software/software_reference/introduction_to_software_history.htm (accessed June 11, 2008).

[19] S. Reiss, *A practical introduction to software design with C++.* New York: John Wiley & Sons, 1998, 397-421.

[20] S. Rakin, "Food for thought: What is software quality assurance?" *Software Quality Consulting* (Jan. 2005, Vol.2 No.1), http://www.swqual.com/newsletter/vol2/no1/vol2no1.html (accessed June 01, 2008).

first program to use IV&V was the Atlas Missile Program in the late 1950s. An independent software tester was hired to conduct unbiased testing of the software. [21]

Over time, the role of IV&V continued to develop and in the 1970's "... the U.S. Army sponsored the first significant such IV&V program for the Safeguard Anti-Ballistic Missile System."[22] The program was designed to identify and eliminate the high risks that are common with military systems. It was successful in meeting its goal and "By the mid- to late 1970's, IV&V was rapidly becoming popular and in some cases was required by the military services..."[23] "It was from this effort that IV&V became well known within the Department of Defense and the aerospace communities as an accepted method of ensuring better quality, performance, and reliability of critical systems."[24]

In the decades following the seventies, IV&V became an intricate part of the software development process. A process that started as "...mostly free-form, not very independent, often started too late to be really effective, and was sometimes even performed by the very people who were developing the system..."[25] grew into process where "...a

---

[21] S. Rakin, "Food for thought: What is software quality assurance?" *Software Quality Consulting* (Jan. 2005, Vol.2 No.1), http://www.swqual.com/newsletter/vol2/no1/vol2no1.html (accessed June 01, 2008).

[22] R. Lewis, *Independent verification & validation: A life cycle engineering process for quality software.* New York: John Wiley & Sons, 1992, xxiii.

[23] Ibid.

[24] Ibid.

[25] R. Lewis, *Independent verification & validation: A life cycle engineering process for quality software.* New York: John Wiley & Sons, (1992): xxiii.

completely independent entity evaluates the work products generated by the team that is designing and/or executing a given project..."[26] The independent entity will also "...monitor and evaluate every aspect of the project itself from inception to completion."[27]

While the cost of conducting IV&V is high, the money saved by preventing errors and rework is far greater. In 1993, the National Aeronautics and Space Administration (NASA) established an IV&V facility in the wake of the Space Shuttle Challenger accident. The facility was developed as part of a plan "to provide the highest achievable levels of safety and cost-effectiveness for mission critical software."[28] "In 2006, NASA allocated $27 Million to the IV&V Facility Budget, of which $19 Million went directly to IV&V Services."[29] After conducting a Return on Investment analysis, "NASA realized a software rework risk reduction benefit of $1.6 Billion in Fiscal Year 2006 alone."[30] From the facilities inception at NASA, it has experienced continued growth while providing better software/system performance, higher confidence in the software reliability, and a reduced maintenance cost.

---

[26] C. Nickolett, "Project due diligence: independent verification and validation." White Paper.Comprehensive Consulting Solutions. Mar 2001: 1-6. http://www.comp-soln.com/IVV_whitepaper.pdf (accessed June 01, 2008).

[27] Ibid.

[28] National Aeronautics and Space Administration (NASA), "NASA IV&V facility - about IV&V." National Aeronautics and Space Administration (NASA), http://www.nasa.gov/centers/ivv/ about/index.html (accessed June 01, 2008).

[29] NASA IV&V Facility, "NASA IV&V 2006 annual report." NASA IV&V Facility, http://www.nasa.gov/centers/ivv/pdf/ 174321main_Annual_Report_06_Final.pdf (accessed June 01, 2008).

[30] Ibid.

When performed correctly IV&V can be a crucial part of the software development process. The process begins with developing Software Integrity Levels (SILs) which "are a range of values that represent software complexity, criticality, risk, safety level, security level, desired performance, reliability, or other project-unique characteristics that define the importance of the software to the user and acquirer."[31] SILs are then used to determine which V&V tasks to perform. The higher the software integrity level, the more V&V tasks assigned. SILs are not constant and can change as software evolves to ensure the appropriate V&V tasks are being performed. Below is an example of SILs based upon the concepts of consequences and mitigation potential as well as an example of V&V processes, activities, and tasks from the IEEE Standard for Verification and Validation. These examples are provided as guidance on how software developers can incorporate IV&V into their software design to assist in reducing specification errors.

[31] Institute of Electrical and Electronic Engineers, Standard for Software Verification and Validation,IEEE-STD-1012, June 08, 2005.

Description of Software integrity Level	Level
Software element must execute correctly or grave consequences (loss of life, loss of system, economic or social loss) will occur. No mitigation is possible.	4
Software element must execute correctly or the intended use (mission) of the system/ software will not be realized, causing serious consequences (permanent injury, major system degradation, economic or social impact). Partial to complete mitigation is possible.	3
Software element must execute correctly or an intended function will not be realized, causing minor consequences. Complete mitigation possible.	2
Software element must execute correctly or intended function will not be realized, causing negligible consequences. Mitigation not required.	1

Figure 1. Examples of Software Integrity Levels

Notes:

1. The V&V tasks may be performed concurrently. The sequential (waterfall) model is shown as an example.
2. V&V tasks listed in this figure are the minimum for Software Integrity Level 4 (Highest Integrity Level).
3. Table 1 contains a complete list of V&V tasks. Table 2 defines the minimum V&V tasks for each software integrity level.

The numbers in parentheses (e.g., 5.1.1) correspond to the subclauses in Clause 5, *Verification and Validation Processes*.

Figure 2. V&V processes, activities, and tasks.

D. CURRENT GUIDANCE FOR VALIDATION

Incorporating IV&V into software design is essential to reducing specification errors. What software engineers need to ensure is when IV&V is applied it is done so correctly. The definitions for V&V provided at the beginning of this chapter allow for the use of computer-based V&V tools to check the correctness of a system or a specific component against a formal specification derived from the natural language requirements. The specifications are created and the final product is then built to satisfy those specifications. Validation that is being conducted in accordance with the guidelines provided by the IEEE evaluates specific components or the final product with the specifications. This process is in fact verification. The product is being built correctly according to the specifications, however, it is not known if the specifications themselves are correct. It is imperative that validation be conducted on the specifications that are created to ensure that the requirements for the project are understood and that the correct product is built.

E. SOFTWARE REUSE

Software reuse is a practice that began in the 1950s with the goal of improving software development productivity and quality. For the past twenty years a great deal of research has been focused on software reuse and its role in software design. Areas that have been given attention include but are not limited to reuse libraries, design patterns, and reuse using formal specifications of requirements. While software reuse holds promise of

improving software development productivity and software quality, the success of reuse is based on the quality of the reusable artifacts. The reuse of software that has not been verified and validated contradicts the intended goal of producing quality software because errors in the software may still exist. This reasoning also holds true when discussing the use of formal requirements specifications. The use of formal requirements specifications is essential in the automation of the software verification process. However, we assert that the correctness of these formal specifications must be first validated before they can be used to verify correctness of the software.

Formal specification has been an active area of research for more than two decades. The requirements specification of a software component describes the expected functions and behavior of the software. The ability to reuse the software component becomes evident if its structure and behavior are compatible with new software being designed.

Verification has been another popular research topic for over 20 years. Automated finite state verification tools have been developed to assist software developers in verifying system specifications. The users of these tools must be capable of specifying the requirements of the system they are developing in the specification language the tool understands. Behavior for a software component is typically specified using temporal logic in an attempt to avoid the ambiguity derived from natural language.

F. FORMAL SPECIFICATION PATTERNS

To assist developers in specifying the behavior in a temporal logic, Dwyer suggests the use of property specification patterns. “A property specification pattern is a generalized description of a commonly occurring requirement on the permissible state/event sequences in a finite state model of a system.”[32] These patterns describe the essential behavior of a system and provide expressions of this behavior in a range of common temporal logics to be used with verification tools. The patterns are then given distinct names describing their behavior which allows them to be mapped to examples of known use, to relationships to other patterns, and to specific formalisms. To facilitate verification, Dwyer proposes the development of a system of property specific patterns for finite state verification tools. The system is a set of patterns or library organized into one or more hierarchies, with connections between related patterns to facilitate the browsing of the system. “A user would search for the appropriate pattern to match the requirement being specified, use the mapping section to obtain the essential structure of the pattern in the formalism used by a particular (verification) tool, and then instantiate that pattern by plugging in the state formula or events specific to the requirement.”[33] The use of these patterns allows for the specification of critical properties

[32] M. Dwyer, G. Avrunin, et.al. “Patterns in property specifications for finite-state verification.” Proceedings of the 21st international conference on software engineering (1999): 411-420.

[33] Ibid.

that exist in software systems and guides users of verification tools to express these properties in a specification language.

In 2005, Konrad and Cheng went a step further with specification patterns and introduced real-time specification patterns that can be used to specify real-time properties for embedded systems. Similar to Dwyer's specification patterns, the real-time specification patterns contain templates for specifying real-time properties in terms of real-time temporal logic.[34] This pattern system is intended to provide strategies for specifying real-time properties in a formal specification language, where the properties are amenable to automated analysis such as verification tools.[35]

Specification patterns and the use of libraries to store those patterns provide another form of software reuse. This form of reuse aims at reducing the cost and improving the quality of formal specification development. However, the effectiveness of the specification pattern reuse depends on the correctness and consistency of the resultant requirements. Proper validation needs to be performed in order to confirm that the requirements are understood.

Otani et al. explains a concept of developing and validating libraries of temporal formal specifications. These libraries would include UML Statechart based assertions for formal specifications and their associated

---

[34] B. Cheng and S. Konrad, "Real-time specification patterns." Proceedings of the 27th international conference on software engineering (2005): 372-381.

[35] Ibid.

validation test scenarios.[36] We intend to build the validation test scenarios with the goal of ensuring that specifications within the libraries are indeed error-free and consistent. The following chapter describes the NASA System Reference Model (SRM) and its role in capturing a modeler's understanding of a specific problem.

[36] T. Otani, D. Drusinsky, et.al. "Validating UML statechart based assertions libraries for improved reliability and assurance." Proceedings of the Second International Conference on Secure System Integration and Reliability Improvement (SSIRI 2008), Yokohama, Japan, July 14-17, 2008, 47-51.

THIS PAGE INTENTIONALLY LEFT BLANK

III. SYSTEM REFERENCE MODEL

A. BACKGROUND

The National Aeronautics and Space Administration (NASA) has continuously developed their IV&V program, supporting new technologies and better validation and verification techniques in an effort to improve the validation and verification process. Earlier versions of the V&V process included the Criticality and Risk Assessment (CARA) and the Software Integrity Level Assessment Process (SILAP). Both processes were found lacking because they relied on manual examination and independent testing of target code. These techniques are ineffective for use in validation because there are no links from the requirements to the system's features, capabilities, properties and functions. Without formal specifications of the system behaviors both CARA and SILAP were unable to validate the correctness and completeness of the developer's understanding of the requirements. Finally, the processes were unable to locate the subtle errors in increasingly complex software-intensive system. Both CARA and SILAP evaluated the risk of software components in a system by compiling a list of software components and evaluating them to prioritize risk assessment, which cannot show that the system being built is the correct system. NASA is in the process of replacing SILAP with advanced computer-aided validation techniques.

The NASA IV&V Facility recognized a need for validation to be more than a risk assessment; it needed to provide a model for the system to show[37]:

- What the system is supposed to do.
- What the system is not supposed to do and
- How the system should respond under adverse conditions.

The NASA IV&V Facility now relies on the use of a System Reference Model (SRM) for each product. "The SRM provides the basis for validating the completeness and correctness of the targeted requirements set."[38] Once the targeted requirements are developed the independent validation team is able to validate those requirements. The SRM supports a computer-aided validation technique through which the independent validation team's understanding and perception of the problem is validated through the team's representation of the SRM's features, properties, function, and capabilities. It is also during this time that the development team is able to discover and correct any identified problems or concerns with their understanding of the requirements for the intended system. This is important because the model holds the responsibility to be complete and accurate to serve its intended purpose and the development team holds the responsibility to ensure that the model fulfills that purpose.

---

[37] K. Woodham, *System Reference Model (SRM) development and analysis guideline,* 1st draft (National Aeronautics and Space Administration (NASA), 2007).

[38] Ibid.

B. DEFINITIONS

The following definitions are described in the SRM guideline[39]. The definition of dependability will be customized to the user's needs and wants of the system.

- Dependability: A dependable system is one that provides the appropriate levels of correctness and robustness in accomplishing its mission while demonstrating the appropriate levels of availability, consistency, reliability, safety, and recoverability.
- Availability: The probability that a system is operating correctly and is ready to perform its desired functions.
- Consistency: The property that invariants will always hold true in the system.
- Correctness: A characteristic of a system that precisely exhibits predictable behavior at all times as defined by the system specifications.
- Reliability: The property that a system can operate continuously without experiencing a failure.
- Robustness: A characteristic of a system that is failure and fault tolerant.
- Safety: The property of avoiding a catastrophic outcome given a system fails to operate correctly.
- Recoverability: The ease for which a failed system can be restored to operational use.

C. SYSTEM REFERENCE MODEL DEVELOPMENT

Without a doubt, any process can become overwhelming in both cost and time. Thus, it is necessary for the SRM to have an appropriate level of specificity so that a completion point can be reached. "The appropriate level of

[39] K. Woodham, *System Reference Model (SRM) development and analysis guideline,* 1st draft (National Aeronautics and Space Administration (NASA), 2007).

V&V is a function of the time available to do the V&V evaluations, and this should in turn be a function of the risk that will be incurred if the V&V is not done, or the risk that will be mitigated if a given level of V&V is done."[40] The SRM still must be developed to a level of fidelity to support validation of the system and result in completeness and correctness of the targeted requirements.

The SRM can be extremely detailed and can consist of high-level use cases, Unified Modeling Language (UML) artifacts such as activity diagrams, sequence diagrams and object class diagrams, and a set of formal assertions to describe precisely the necessary behaviors to satisfy system goals, with respect to the three questions stated previously. These many artifacts allow the team to properly express the requirements through the SRM and ensure that their understanding of the requirements is correct.

The development of the SRM begins with a scoping period. During this time the SRM development team commences with a front-end analysis. The front-end analysis ensures that the team has a clear perspective of the intended use of the model. This high-level abstraction helps the team ensure that the model is defined which in-turn drives the objectives of the model development. The scoping period also ensures that the SRM development is based on concept-level documentation rather than requirements generated by the

[40] R. Logan and C. Nitta, "Verification & validation: process and levels leading to qualitative or quantitative validation statements." *SAE Transactions* vol.113, no.5 (2004), http://bill.cacr.caltech.edu /valworkshop/upload/files/UCRLTR-200131sae04fa.pdf (accessed June 01, 2008).

system developers. Finally, the scoping period should finalize the level of specificity of the requirements so that a completion point can be reached.

The scoping period consists of analyzing the constraints, restrictions and targeted tasks and requirements to recognize the depth of the modeling needed. Additionally, requirements that will not be modeled in the SRM are identified and the team ensures that sufficient concept documentation is available to continue. The concept documents used during the process are found in many forms of stakeholder inputs from mission statements to concepts of operations. The scoping period ends with a clear understanding of the system elements that need to be addressed and the depth that they need to be defined. The level of fidelity should be determined at this point to ensure completeness and correctness of the targeted system requirements.

The next stages of the SRM development are accomplished through the development of use cases and UML artifacts as well as supporting assertions. The SRM team, using the conceptual documents, will begin by documenting system behaviors. It is during this time that the system goals should be identified and a traceability matrix developed, populated with these top-level goals. Additionally, the operational environment must be identified and the traceability matrix should be populated with operation environment characteristics that need to be addressed by the system model. The top-level use cases developed to address the overall system goals are peer reviewed to validate that the preliminary use case set spans the high-level

description of the system and is documented in the traceability matrix. The top-level use cases are abstracted from the details of the system and are goal-oriented. These use cases help the developers to get a clear understanding of the process and problems to be solved as well as the goals and objectives of the system. The top-level use cases then are refined into lower-level use cases and activity diagrams which can be mapped to sequence diagrams. The process continues to become more specific to ensure that the goals and objectives are accomplished but also to verify that their constraints are also adequately captured. The diagrams should provide a complete representation of the behavior expected to be displayed by the system. Additionally, all behaviors should be mapped and defined into the traceability matrix and peered reviewed to ensure correctness. The overall goal is to ensure that the top-level use cases have been refined into detailed lower-level uses cases that represent not only the Main Success Scenario (MSS) but are fully elaborated to ensure necessary extensions are also represented. Finally, the modeling team has to ensure that any dependability considerations are addressed and represented in the model. This entire effort should represent the desired system behaviors as well as any necessary extensions and assertions that map to the top-level goals and requirements. The model is ready to be validated.

D. VALIDATING THE SYSTEM REFERENCE MODEL

The newly developed SRM is a representation created by the SRM development team and is a result of the teams own perceptions and understanding of the desired system

behaviors. As such the representation could be wrong if the team misinterpreted the desired behaviors of the system. This is why the SRM must be validated to reduce specification error as well as to ensure that the behavior requirements created by the SRM development team are measured against the SRM for correctness. The SRM is a model of the intended system and it must meet any dependable considerations in order for the intended system to be so as well.

The validation process is twofold and can begin with a formal review and tracing of the UML artifacts to include: use case definitions and models, supporting assertions, and activity diagrams. Other artifacts reviewed include the complete set of system-behavior definitions based on stakeholder goals and system constraints and operations environments defined in the concept documentation. During this review the formal tracing of the requirements from the top-level to the more refined lower-levels and the activity diagrams and sequence diagrams helps to identify the subsystems and components responsible for the system requirements. Additionally, during this process all the requirements are elicited and peer reviewed. This ensures that all targeted requirements have been identified and traced through the artifacts. During this time all necessary objects and events are labeled, identified, and checked to ensure there are no unnecessary objects or events. The above process ensures that all targeted requirements are fully detailed in accordance with their goals. During each review each step is subjected to extensive group review to validate that the SRM is a complete and unambiguous representation of the system.

The second step of the validation process is to execute as much of the model as possible through computer-aided auditing. Run-time verification of formal assertions is able to check for inconsistency, omission and errors in the SRM. By executing as much of the model as possible it increases the evidence that the model being developed is the correct system. The independent validation team is able to use the evidence of validation to ensure that the SRM is the correct system.

The IV&V team’s requirements elicitation and validation tasks produce deliverable packages, consisting of: UML models for reference model constituents, natural language assertions, formal representation of the assertions, and a validation test suite for each assertion. The test suites are detailed and include tests that cover multiple scenarios that meet the requirements of the assertions, and will be discussed further in the next chapter. These deliverable packages are the evidence gathered to decrease specification errors and must be done to validate the SRM and provide evidence of dependability of the system.

The SRM is intricate and detailed in order to show its dependability. But before dependability can be shown the SRM assertions must be validated to decrease specification errors. In fact the assertions should precisely model the required behavior of the system and if they are able to do so the model is on its way to being validated. But assertions also have to be validated and are validated through an execution-based model checker for dependability of the model under test.

E. INCREASING THE USABILITY OF THE SRM

The difficulty with assertions is in their creation. It not only takes time and effort, but the correctness of the executable assertions depends on the ability of the modelers to specify correct assertions. It is difficult to specify and develop correct assertions. The modelers must have a correct representation of the structure and behavior of the SRM, the assertions must also be correct. If faulty assertions are used they are not effective in the IV&V process. We believe that a library built with correct assertions would enable the assertions to be reused. This could both decrease the burden on the modelers to develop the assertions and improve the ability of the independent validation teams to validate the dependability of the software.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. BUILDING AN ASSERTION LIBRARY

A. BACKGROUND

As mentioned in the previous chapters the SRM is a representation created by the developers as a result of their own understanding of the desired system behaviors. The SRM must be validated to reduce specification errors. One of the ways to do this is through assertions which precisely model the required behavior of the system and are the foundation of the SRM. Through testing and modeling assertions the independent validation team begins to comprehend the problem domain and refine any problems to ensure that the SRM meets the user's requirements and the correct system is built. The current way to build assertions is to develop the assertions from natural a language description of the user's understanding anew every time; this can be a time-consuming and error-prone undertaking. We believe that an assertion library can help ease these tasks by providing validated assertions which can be reused.

The purpose of this chapter was to construct an approach to building an assertion library with a small number of assertions that have been validated for correctness and are reusable. We define a library to be a collection of assertions that are stored, collectively shared and can be filled with more assertions as needed. The assertions in the library are validated through the use of test scenarios that we designed. The test scenarios are patterns which test the assertion for the required behavior. The purpose of the test suites is to disambiguate the

assertions, and test for correctness meaning that the assertions accurately reflect the natural language statement as we intended.

The assertion library would be built so that the assertions are reusable and adaptable for future projects. Software developers can select from the library any assertions that meet their needs and adapt them or use them as an example to build their own. In each case we ensured that the assertion was general to increase the ability to be reused as well as be more relevant to the library. We hope that through this process that software developers will be able to use our correct assertions in the library for their own use and reuse, lessening their burden and reducing specification errors in the software.

B. STATECHART ASSERTIONS

The libraries are built through the use of "UML statechart based temporal assertions for formal specifications."[41] The UML statecharts are developed from both the research efforts of Harel, who first proposed the use of statechart diagrams as a visual approach to modeling the behavior of complex reactive system, and Drusinsky who both increased and extended the use of statechart diagrams to specify formal assertions. Drusinsky was able to extend the use of statecharts as formal assertions for temporal behavior with "the inclusion of a built-in Boolean flag bSuccess and a corresponding isSuccess method which specifies the Boolean status of the assertion true if the

[41] D. Harel. *Statecharts: A visual approach for complex systems, Science of Computer Programming*, vol.8, no.3. (1987): 231-274.

assertion succeeds and false otherwise."[42] The statechart assertion indicates that "formalism is supported by StateRover, a design entry, code generation, and visual debug animation tool for UML statecharts combined with flowcharts."[43] Assertion statecharts can be nondeterministic and deterministic depending on the needs and wants of the developer and modeler. For example, the developer might want a nondetermistic statechart if there are nested requirements which can be more difficult to write and less readable in a deterministic solution. Or alternatively if the assertion needs to be active in runtime, then a deterministic statechart might be a better solution because of the overhead incurred in the nondeterministic statechart at runtime.

Finally, it is important to understand the proper use of a statechart assertion. Remember that the assertion uses the "built-in Boolean variable name bSuccess, and a corresponding method called isSuccess(), both automatically created by the code generator"[44] to make a statement about the assertion's correctness. The default settings of the assertion statechart variable bSuccess is set to true. To appropriately test success and failure, the modeler needs to

---

[42] D. Drusinsky. *Modeling and verification using UML statecharts - a working guide to reactive system design, runtime monitoring and execution-based model checking.* Elsevier Inc., 2006.

[43] D. Drusinsky, M. Shing, K. Demir, "Creation and validation of embedded assertion statecharts", *Proc. 15th IEEE International Workshop in Rapid System Prototyping*, Greece (June 14-16, 2006): 17-23

[44] D. Drusinsky. *Modeling and verification using UML statecharts - a working guide to reactive system design, runtime monitoring and execution-based model checking.* Elsevier Inc., 2006.

ensure that the assertion enters the error state and the on-entry action assigns bSuccess=false when the assertion fails.

C. ASSERTION VALIDATION

Once the natural language has been translated into an assertion the assertion must then be validated. The assumptions in the statechart must be tested to ensure that the statechart assertion correctly represents the intended behavior the modeler has in mind. We need to run validation test scenarios against the statechart assertion.

In each case the validation test suite resolved the ambiguities of the natural language specification. The tests were meaningful in that they ensured each assertion were distinguishable from each other. The assertions were tested and we did find that, when we tested them, we had to disambiguate the natural language ourselves to ensure that we truly understood what we were describing.

The two kinds of errors that are commonly found were "implementation errors resulting from mistakes in the statechart assertion, and errors or ambiguities in the natural language statement." [45] In the first case, the statechart behavior does not match the modeler's intended behavior. The second case was more difficult because it depended how we as individuals understood the natural language statement and how we as individuals clarified the

[45] T. Otani, D. Drusinsky, et.al. "Validating UML statechart based assertions libraries for improved reliability and assurance." Proceedings of the Second International Conference on Secure System Integration and Reliability Improvement (SSIRI 2008), Yokohama, Japan (July 14-17, 2008): 47-51.

[45] Ibid.

ambiguities. It was by running the test scenarios that we were able to identify these errors and modify our assumptions and assertions accordingly in order to correct the assertions.

Otani et al.[46] revealed that there are some types of patterns that must be part of every validation test-suite:

- Obvious success. We expect that the statechart assertion being validated to succeed on such a test.
- Obvious failure. We expect that the statechart assertion being validated to be violated on such a test.
- Event repetitions. We create event repetitions and assure that the assertion, if applicable, is not written in a manner that only observes the first occurrence of a triggering event P in a sequence of P's.
- Multiple time intervals. If the assertion requires it, we check that it handles multiple time intervals or scenarios. By using this validation test pattern we assure that an assertion is not written in a manner that observes only a single time interval.
- Overlapping time intervals. If the assertion requires it, we check that the assertion can handle overlapping time intervals within a scenario.

Once the types of patterns were clarified we then designed our test suite to adhere to the above categories, combining them if suitable and ensured that there were an appropriate number of tests per test suite that would validate the assertion.

[46] T. Otani, D. Drusinsky, et.al. "Validating UML Statechart Based Assertions Libraries for Improved Reliability and Assurance." Proceedings of the Second International Conference on Secure System Integration and Reliability Improvement (SSIRI 2008), Yokohama, Japan, (July 14-17, 2008): 47-51.

D. ASSERTION SCENARIOS

The first assertion statement that we described is: Whenever P then Q within T. The assertion statechart (shown in Figure 3) was diagramed as follows:

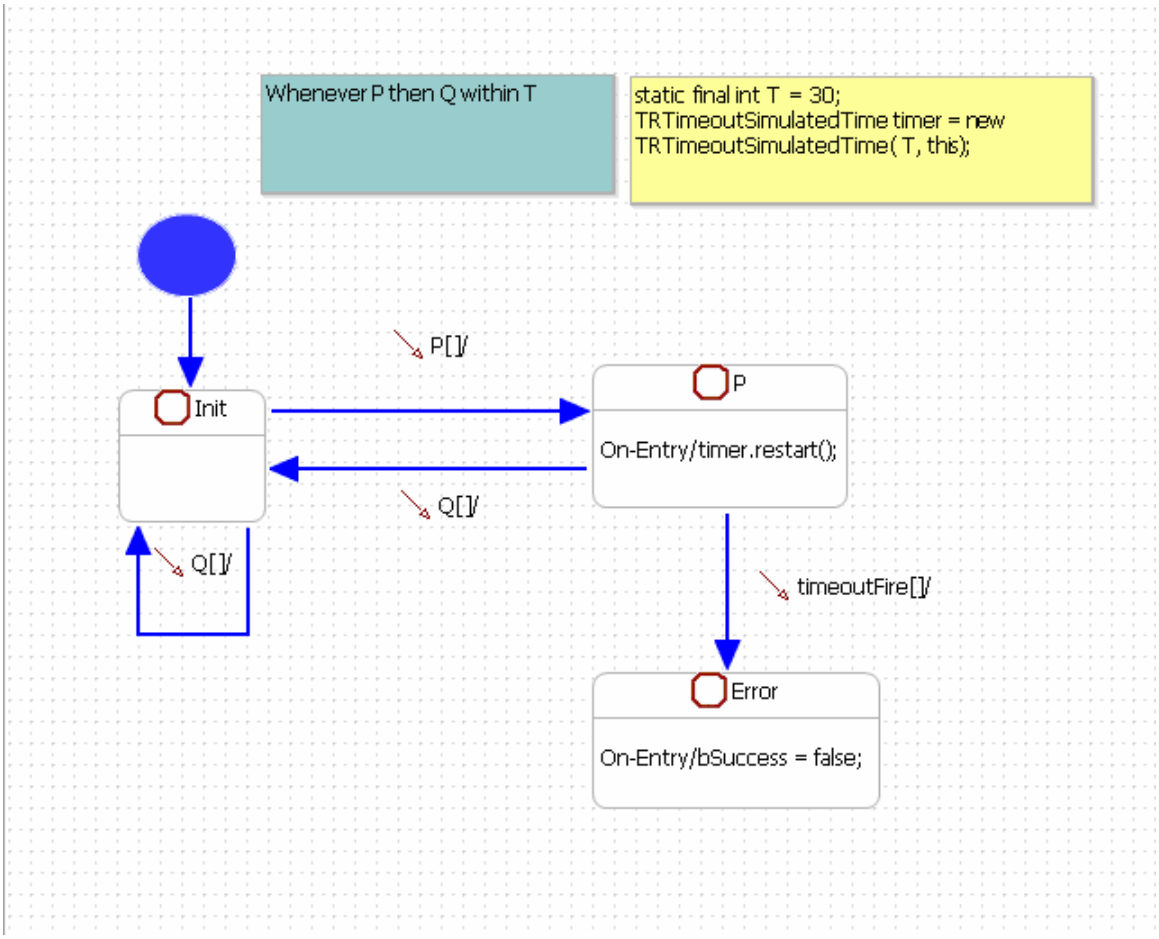
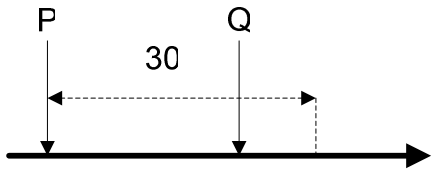


Figure 3. Whenever P then Q within T

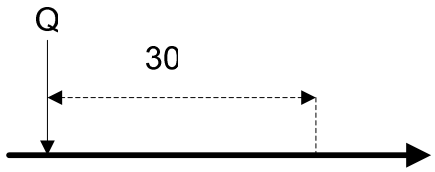
Our interpretation of the assertion statement above is: if P occurs (timer is reset at every P) then the event Q will eventually occur within the time interval. The built in event, timeoutFire(), fires after 30 sec. In case of a P repetition before a Q the 30 second duration will be measured from the first p. We used the following patterns to

correctly disambiguate the natural language and ensure that the assertion statement accurately reflects the natural language as we identified and desired. As described earlier in the chapter we covered all appropriate testing patterns.

Obvious success:

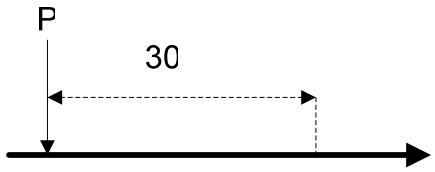


P; incrTime(25); Q; incrTime(6)(timeout has occurred). We expected this test to be a success. Our expectation was confirmed.



Q; incrTime(31) (timeout has occurred). We expected this test to be a success because we are testing for violations of the assertion and Q by itself does not violate the assertion. Our expectation was confirmed.

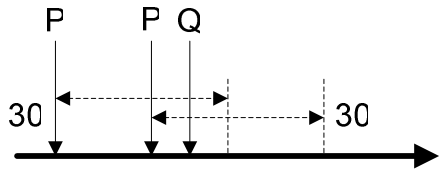
Obvious failure:



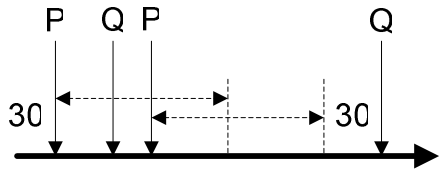
P; incrTime(31) (timeout has occurred). We expected this test to fail because it did not meet the constraints of the assertion. Our expectation was confirmed.

Overlapping time intervals:

In this test and the next test we ensure that the assertion observes more than the first P in a sequence of P’s.



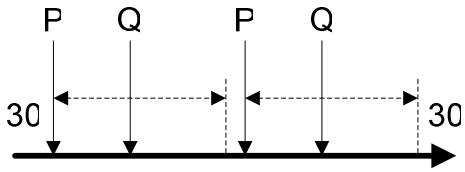
P; incrTime(15); P; incrTime(5); Q; incrTime(26) (timeout has occurred). Our goal in this test was to ensure that the assertion could handle overlapping time intervals. We expected success. Our expectation was confirmed.



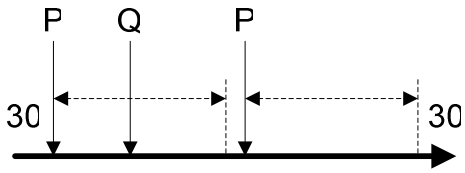
P; incrTime(5); Q; P; incrTime(31) (timeout has occurred); Q. Our goal in this test was to test overlapping time intervals for an expected failure as this test does not meet the constraints of the assertion. Our expectation was confirmed.

Multiple Intervals:

We tested for multiple intervals in this test and the next to ensure that the assertion would observe more than a single time interval.



P; incrTime(10); Q; incrTime(20); P; incrTime(10); Q; incrTime(21) (timeout has occurred). We expected success because it meets the requirements of the assertion. We set bSuccess = true. Our expectation was confirmed.



P; incrTime(10); Q; incrTime(20); P; incrTime(31) (timeout has occurred). We tested for multiple intervals expecting failure because of the constraints of the assertion. Our expectation was confirmed.

The second core assertion statement that we described was: Whenever P then no Q within T. The assertion statechart (as shown in Figure 4) was diagramed as follows:

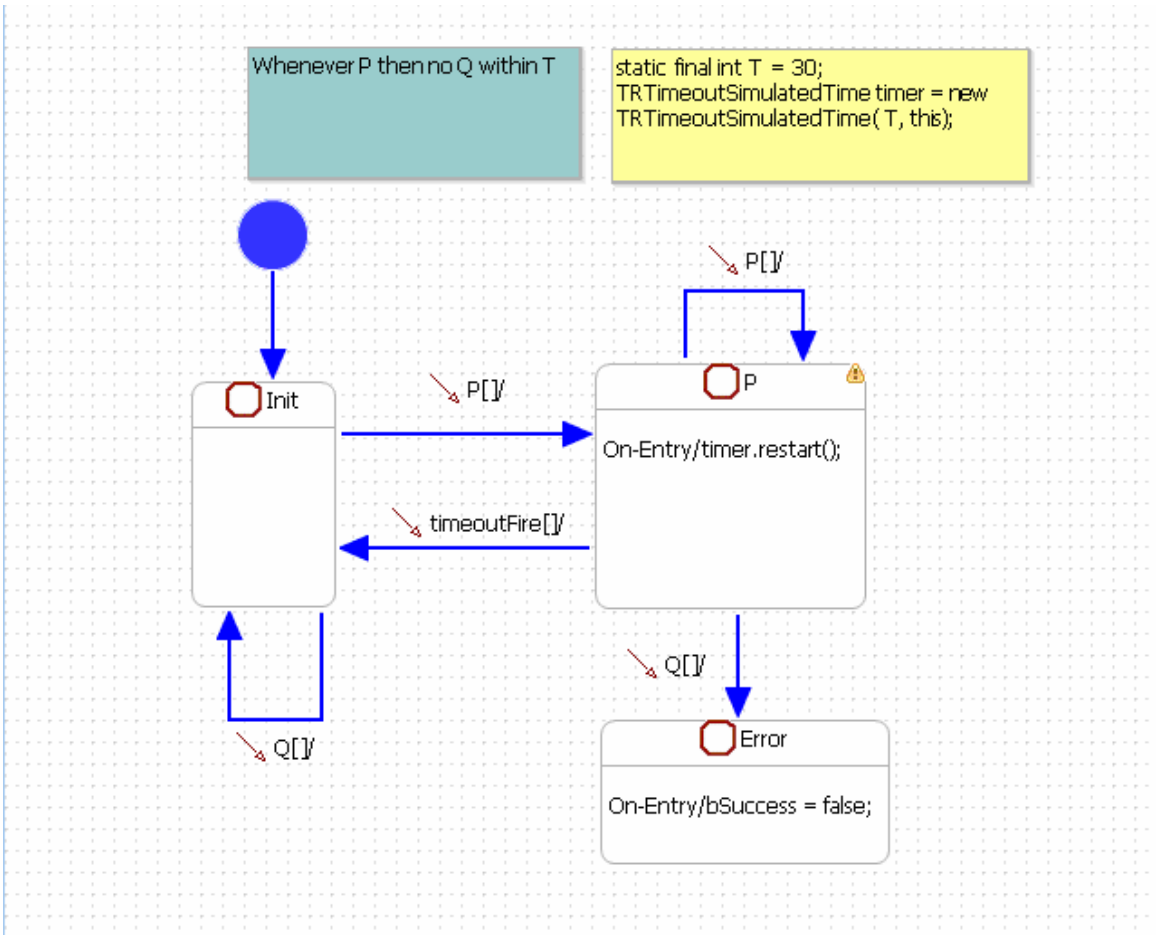
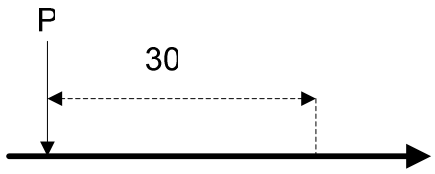


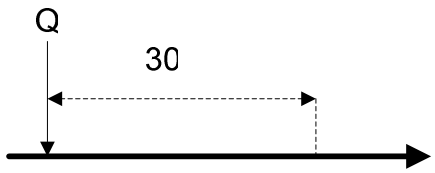
Figure 4. Whenever P then no Q within T

Our interpretation of the assertion is if P, then within the time interval for P no Q will appear. The built in event, timeoutFire(), fires after 30 sec. A P repetition would reset the timer. We used the following patterns to disambiguate the assertion statement.

Obvious success:

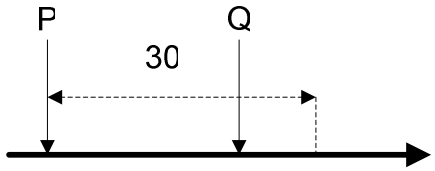


P; incrTime(31) (timeout has occurred). We expected this test to be a success because no Q occurred which meets the requirements of our assertion. Our expectation was confirmed.



Q; incrTime(31) (timeout has occurred). We expected this test to be a success because we are testing for violations of the assertion and Q by itself does not violate the assertion. Our expectation was confirmed.

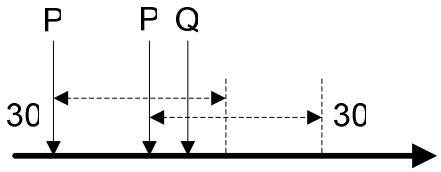
Obvious failure:



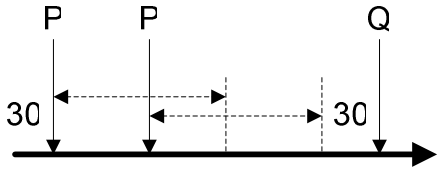
P; incrTime(25); Q; incrTime(6) (timeout has occurred). This test was expected to be a failure because it violates the requirements of the assertion. Our expectation was confirmed.

Overlapping time intervals:

In this test and the next test we ensure that the assertion observes more than the first P in a sequence of Ps.



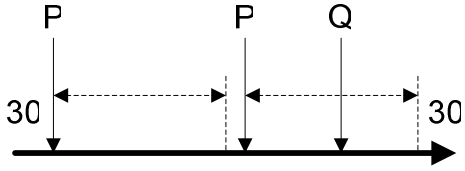
P; incrTime(15); P; incrTime (5); Q; incrTime(26) (timeout has occurred). Our goal in this test was to ensure that the assertion could handle overlapping time intervals. The test was expected to be a failure. Our expectation was confirmed.



P; incrTime(10); P; incrTime(31)(timeout has occurred); Q. The test was expected to be a success because Q was not injected during the P intervals. Our expectation was confirmed.

Multiple Intervals:

We tested for multiple intervals in this test to ensure that the assertion would observe more than a single time interval.



P; incrTime(30); P; incrTime(15); Q; incrTime(16) (timeout has occurred). This test was expected to be a failure because it does not meet the constraints of the assertion. Our expectation was confirmed.

E. CONCLUSION

When we first defined the assertions in natural language we discovered that almost all assertions can be ambiguous and difficult to define at first. The natural language statements meant different things to different people. “If P then Q within T” could mean an interval T measured from the first or the last P depending on how it was defined and what the software developers wants to test. We disambiguated each assertion according to the most general and useful definition; this meant that in most cases the assertion would be general and not specific so as to be more useful. There was additional difficulty as can be expected with any new system as StateRover is still in development. But we were able to succeed after several restarts and debugging help. Finally, during our disambiguation period we fell victim to the statechart default which is bSuccess = true. During the testing period we expected one result and received something completely different. This led us to additional testing and clarification of the assertions and we had to ensure every time that the assertion test was not successful because the bSuccess flag was set to true, but rather because the test was actually correct.

This process is incredibly interesting and requires clarity of thinking as well as the ability to break down natural language. It is not simple but the process invokes greater understanding of the validation process and the validity of the assertion library. We feel that these assertion statements can be built upon and reused for the benefit of validation purposes.

Additional assertion statecharts and validation test suites that we defined and tested can be found in Appendix A. A final assertion statechart and validation test suite that has merit but is not as valuable as previous mentioned assertions can be found in Appendix B.

V. CONCLUSION

A. SUMMARY AND CONTRIBUTIONS

Software has become a vital part of our everyday lives. Whether we refer to our military systems, medical systems, or our financial systems, software is a part of them and has become something that we now depend on. In our thesis we concentrate on requirements and their formal specification, and we discuss a method to reduce specification errors. We strive to find a better technique to answer the question “Are we building the right product?” Validation presents a means of assuring that software satisfies the user’s requirements. It is viewed as a way of saving time and money that could otherwise be wasted if a product design is not built correctly and rework needs to be conducted. A problem that can exist when conducting validation is not conducting validation early enough in the design process. Often the user’s requirements are reviewed and specifications are developed. The product design is then built according to the specifications. Once the product design is built validation is conducted by comparing the resultant product with the original requirements. As software partition of systems continues to grow and become more complex we assert that validating a product after it is developed is too late in the process. At that stage the amount of time and cost of rework that may need to be performed is too large. Validation needs to begin earlier in the design process by ensuring the specifications themselves are correct and consistent.

At present several ways to conduct validation exist. Some guidance that is provided actually describes verification when referring to validation by having the product design compared to the specifications for the project. Others suggest what we have already mentioned and that is to check the final product against the user requirements. To conduct validation we introduced a process of developing and validating temporal formal specifications in the form of statechart assertions. Included in our work are validation test scenarios intended to ensure specifications are in fact correct prior to moving forward with a project. The goal is to make available multiple libraries of pre-vetted assertions to facilitate validation.

This research described the attributes of IV&V as well as software reuse and explores a concept of combining the process of validation and reuse in an attempt to yield a repeatable validation technique. Sample requirements were identified and then formal specifications in the form of statechart assertions were created to capture the requirements. Testing scenarios were then developed to determine if the statechart assertions were accurate and consistent with the original requirements. Once these assertions are proven to be accurate they can be stored in a library for future reuse. Our intentions are to ensure that specifications used to build a product are validated prior to time and money going into building the final product. By using an assertion repository filled with correct assertions to build the specifications for a design, the engineer can be sure that the specifications used to build the final product are correct. If errors are found in the specifications the engineer can go back and find out where

the error is coming from. This would be faster and cheaper than correcting software that has already been developed in accordance with incorrect specifications.

B. FUTURE WORK

The goal in both the DoD and the software industry is to produce software that is cost effective, reliable, maintainable, and above all usable. The current guidance on verification and validation that exists does not provide a technique to show engineers how to create software that possesses these attributes. The guidance that does exist leaves software engineers to develop their own verification and validation methods.

The amount of work that could be conducted in the software industry to ensure reliable software is being produced is abundant. We have established an approach for developing and validating statechart assertions as a road map to produce reliable software. One avenue of future work would be to further expand this approach by developing additional assertions that apply to a specific domain. For example, select a domain of interest such as theatre ballistic missile defense. Then, determine requirements that exist within that domain. Once the requirements for the specific domain are understood, translate the natural language of the requirements into assertions as we did in chapter IV. Then validate the assertions through the use of test cases to ensure that the statechart assertions correctly represent the intended behavior the modeler has in mind.

Another avenue of future work would be to create a library to store the assertions in. When creating the library the developer will need to consider the size of the library and how many assertions will be placed in it. The developer will need to decide if several libraries are to be developed to categorize the different assertions or if the assertions will be organized within one library in a manner that will be easy to search. Once the organization of the library is decided information retrieval will need to be focused on. How will assertions be retrieved or called from the library? What will be the best interface to facilitate information retrieval and the use of the assertions? The goal should be to find an acceptable interface that does not cause errors of its own. Another area to look at is the adaptation of the assertions to a library environment. Do they perform as expected? One goal the developer should seek is to automate the processes of organizing, retrieving information from, and interfacing the libraries as much as possible in an attempt to reduce errors.

Finally, once a library is developed, a future project could focus on how to best maintain that library to facilitate future use. One item to consider is if certain assertions are used more frequently than others. In this case the developer would want to set up the library in a way that the frequently used assertions can be searched before the rest of the library is searched. A way to enable this would be to maintain a count of how often each assertion is used. Also if an assertion is proven not to be used, a way to comment the assertion out in the library to eliminate it from future searches may prove useful. Doing this may be a way to enable faster searches thereby saving time in the

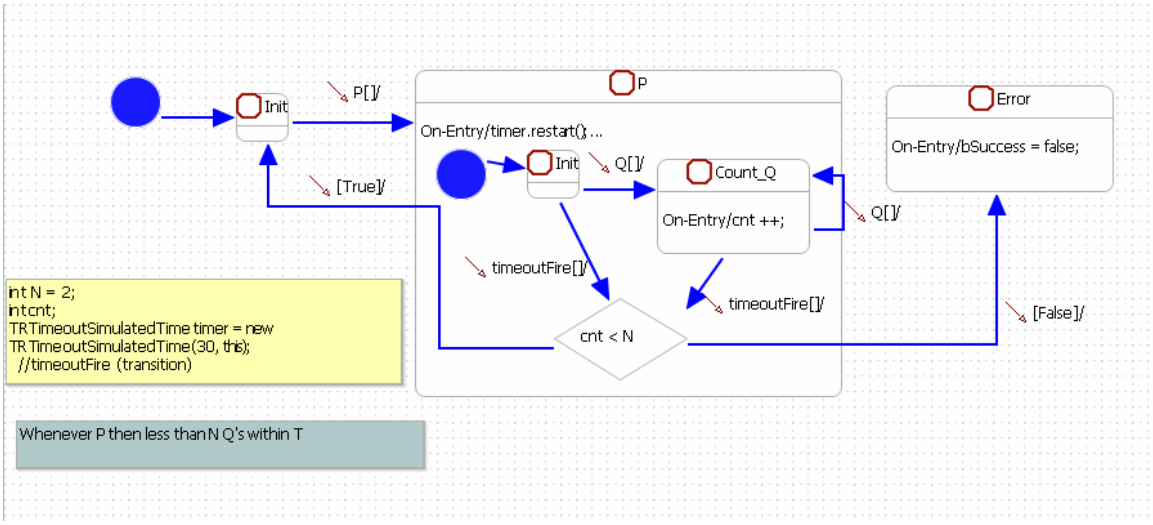
development process. By commenting the assertion out rather than removing it from the library it can still be included in future searches if it is decided that it is needed.

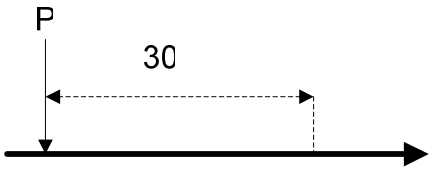
THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX: ADDITIONAL ASSERTION DIAGRAMS AND TEST SUITES

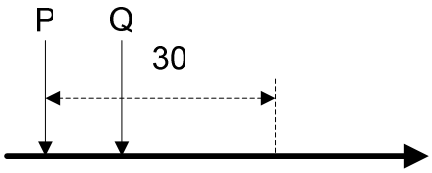
A. ADDITIONAL ASSERTION DIAGRAMS BOUNDED BY TIME

1. Whenever P Then Less Than N Qs Within T

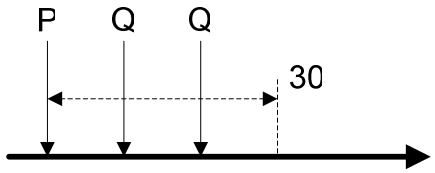




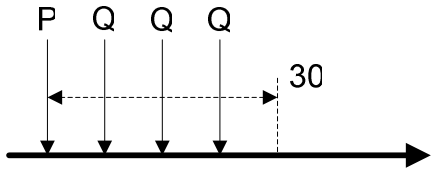
P; incrTime(31) (timeout has occurred). We expected an obvious success. Our expectation was confirmed.



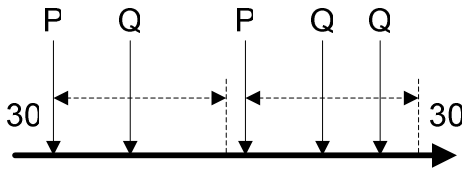
P; incrTime(5); Q; incrTime(26) (timeout has occurred). We expected an obvious success. Our expectation was confirmed.



P; incrTime(5); Q; incrTime(5); Q; incrTime(21) (timeout has occurred). We expected failure since we set N to 2. Our expectation was confirmed.

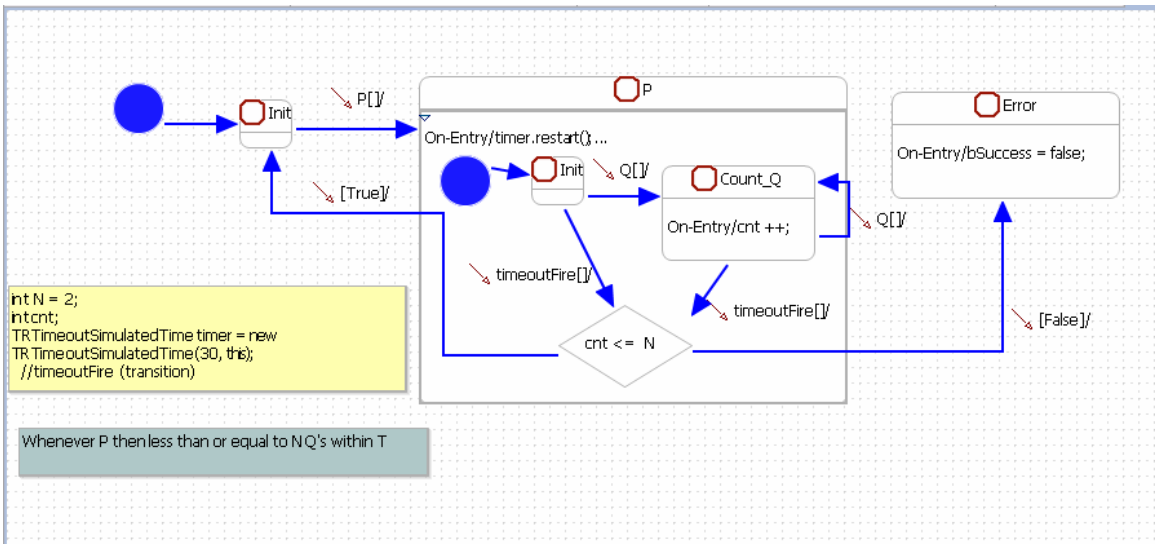


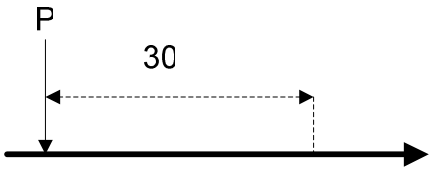
P; incrTime(5); Q; incrTime(5); Q; incrTime(5); Q ; incrTime(16)(timeout has occurred). We expected failure since we set N to 2. Our expectation was confirmed.



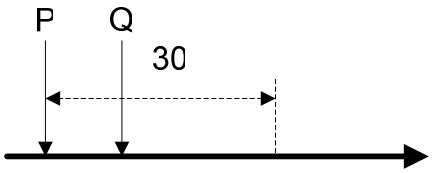
P; incrTime(5); Q; incrTime(26)(timeout has occurred); P; incrTime(5); Q; incrTime(5) Q; incrTime(21) (timeout has occurred). We tested for multiple intervals in this test to ensure that the assertion would observe more than a single time interval. We expected failure in the second interval since we set N to 2. Our expectation was confirmed.

2. Whenever P Then Less Than or Equal to N Qs Within T

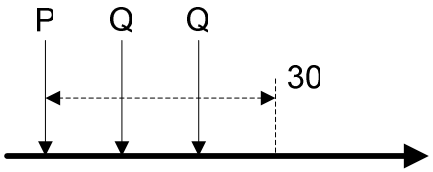




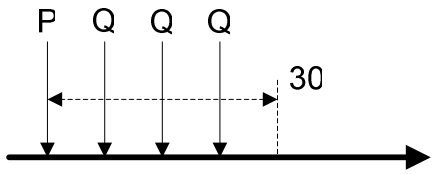
P; incrTime(31) (timeout has occurred). We expected an obvious success. Our expectation was confirmed.



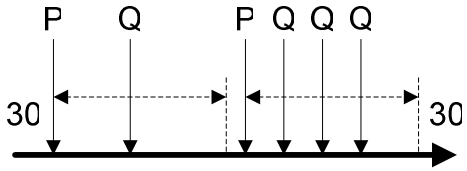
P; incrTime(5); Q; incrTime(26) (timeout has occurred). We expected an obvious success. Our expectation was confirmed.



P; incrTime(5); Q; incrTime(5); Q; incrTime(21) (timeout has occurred). We expected an obvious success. Our expectation was confirmed.

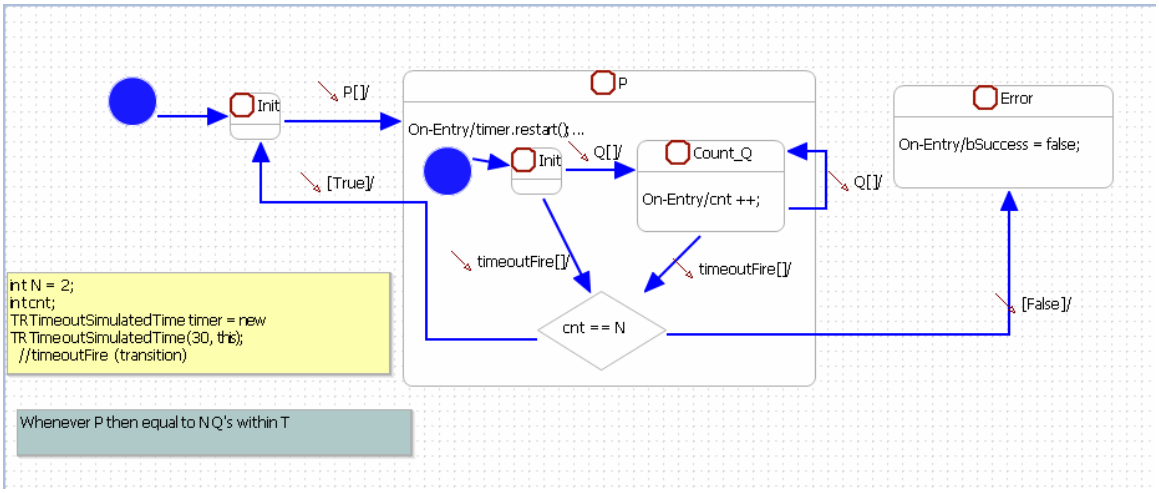


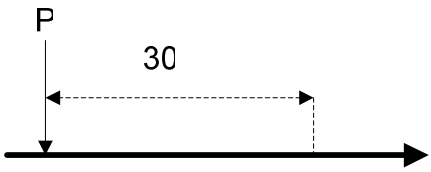
P; incrTime(5); Q; incrTime(5); Q; incrTime(5); Q ; incrTime(16)(timeout has occurred). We expected obvious failure since we set N to 2. Our expectation was confirmed.



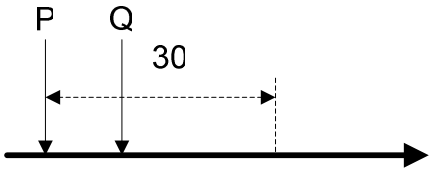
P; incrTime(5); Q; incrTime(26)(timeout has occurred); P; incrTime(5); Q; incrTime(5) Q; incrTime(5); Q; incrTime(16) (timeout has occurred). We tested for multiple intervals in this test to ensure that the assertion would observe more than a single time interval. We expected failure in the second interval since we set N to 2. Our expectation was confirmed.

3. Whenever P Then Equal to N Qs Within T

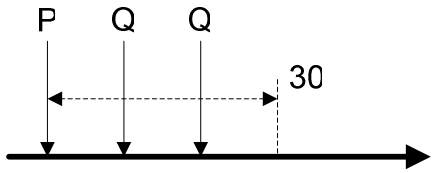




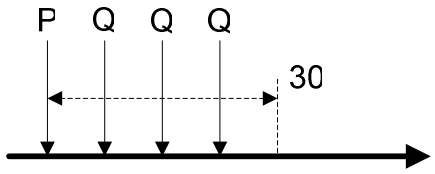
P; incrTime(31) (timeout has occurred). We expected obvious failure since we set N to 2. Our expectation was confirmed.



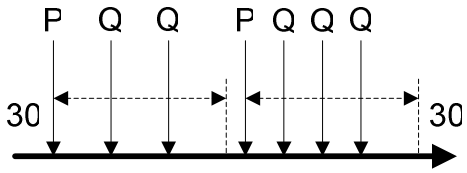
P; incrTime(5); Q; incrTime(26) (timeout has occurred). We expected obvious failure since we set N to 2. Our expectation was confirmed.



P; incrTime(5); Q; incrTime(5); Q; incrTime(21) (timeout has occurred). We expected obvious success since we set N to 2. Our expectation was confirmed.

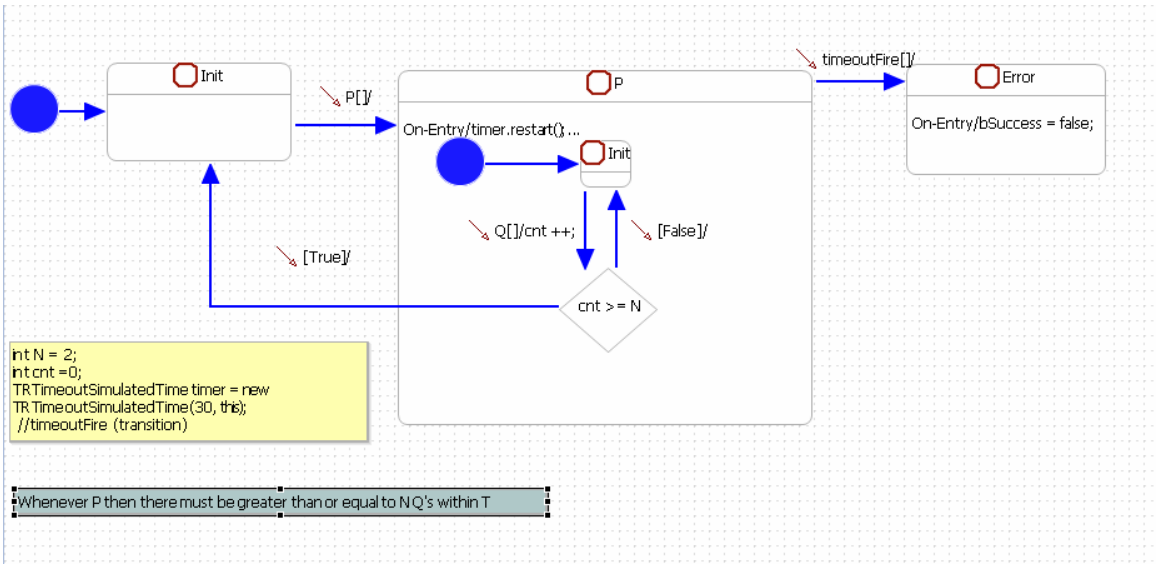


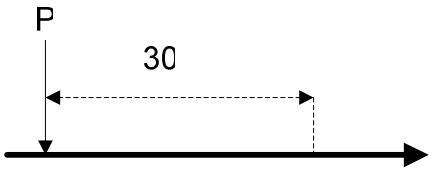
P; incrTime(5); Q; incrTime(5); Q; incrTime(5); Q ; incrTime(16)(timeout has occurred). We expected obvious failure since we set N to 2. Our expectation was confirmed.



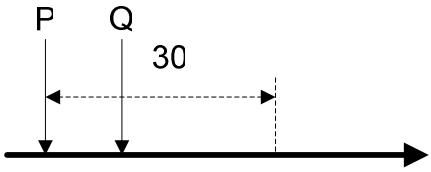
P; incrTime(5); Q; incrTime(5); Q; incrTime(21)(timeout has occurred); P; incrTime(5); Q; incrTime(5) Q; incrTime(5); Q; incrTime(16) (timeout has occurred). We tested for multiple intervals in this test to ensure that the assertion would observe more than a single time interval. We expected failure. Our expectation was confirmed.

4. Whenever P Then Greater Than or Equal to N Qs Within T

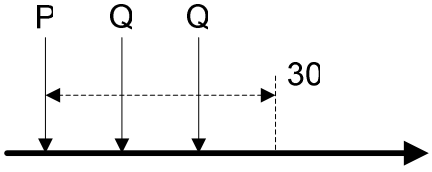




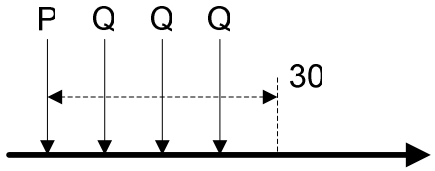
P; increment time to 31; timeout. We expected obvious failure since we set N to 2. Our expectation was confirmed.



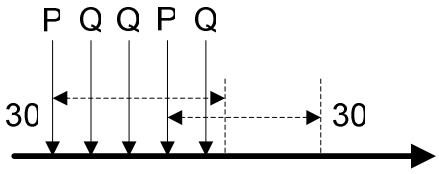
P; incrTime(5); Q; incrTime(26) (timeout has occurred). We expected failure since we set N to 2. Our expectation was confirmed.



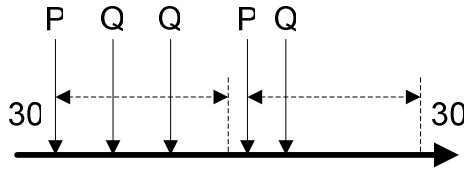
P; incrTime(5); Q; incrTime(5); Q; incrTime(21) (timeout has occurred). We expected success since we set N to 2. Our expectation was confirmed.



P; incrTime(5); Q; incrTime(5); Q; incrTime(5); Q ; incrTime(16)(timeout has occurred). We expected success since we set N to 2. Our expectation was confirmed.

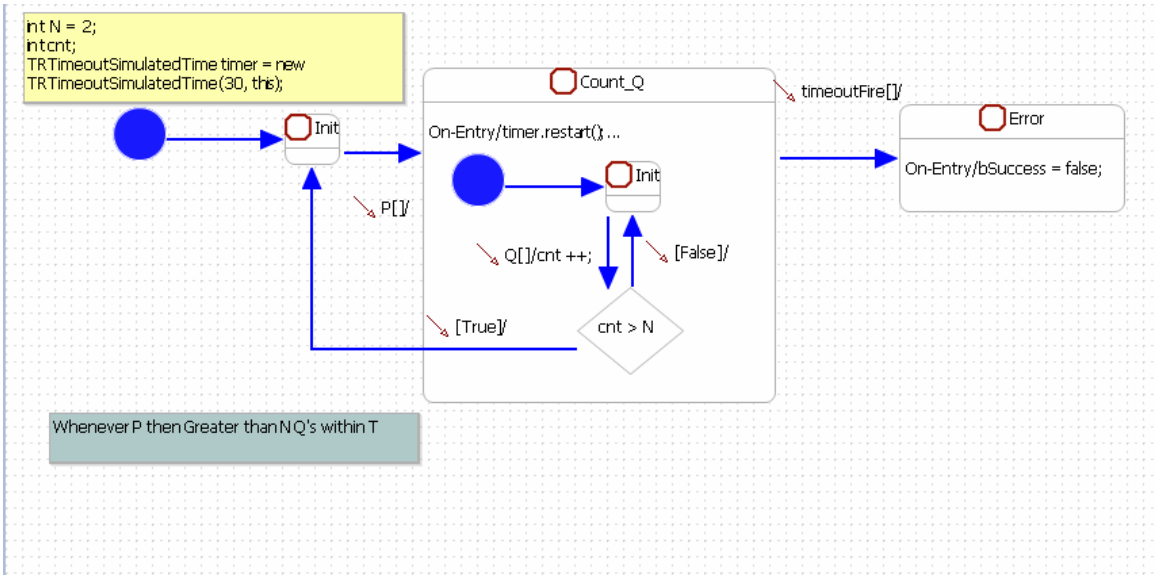


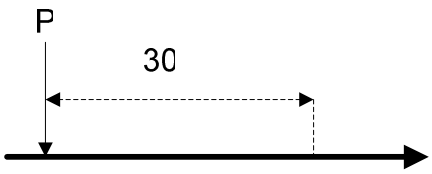
P; incrTime(5); Q; incrTime(5); Q; incrTime(5) P; incrTime (5); Q; incrTime(26)(timeout has occurred). Our goal in this test was to ensure that the assertion could handle overlapping time intervals and that the assertion observes more than the first P in a sequence of Ps. The test was expected to be a failure since we set N to 2 which violates the second P. Our expectation was confirmed.



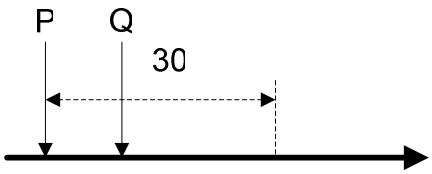
P; incrTime(5); Q; incrTime(5); Q; incrTime(21)(timeout has occurred); P; incrTime(5); Q; incrTime(26) (timeout has occurred). We tested for multiple intervals in this test to ensure that the assertion would observe more than a single time interval. We expected failure since we set N to 2. Our expectation was confirmed.

5. Whenever P Then Greater Than N Qs Within T

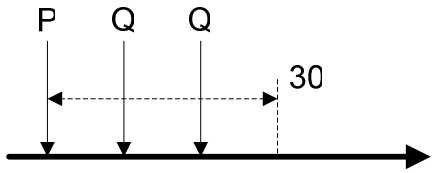




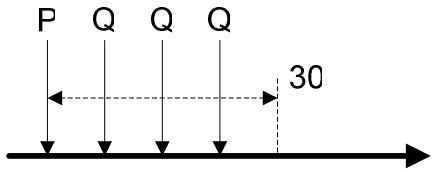
P; incrTime(31)(timeout has occurred). We expected failure since we set N to 2. Our expectation was confirmed.



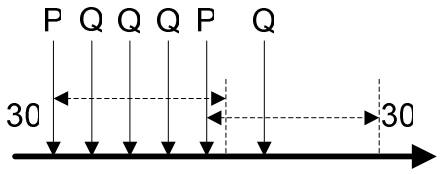
P; incrTime(5); Q; incrTime(26) (timeout has occurred). We expected failure since we set N to 2. Our expectation was confirmed.



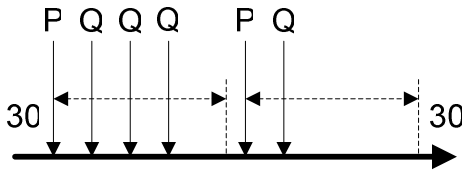
P; incrTime(5); Q; incrTime(5); Q; incrTime(21) (timeout has occurred). We expected failure since we set N to 2. Our expectation was confirmed.



P; incrTime(5); Q; incrTime(5); Q; incrTime(5); Q; incrTime(31)(timeout has occurred). We expected success since we set N to 2. Our expectation was confirmed.

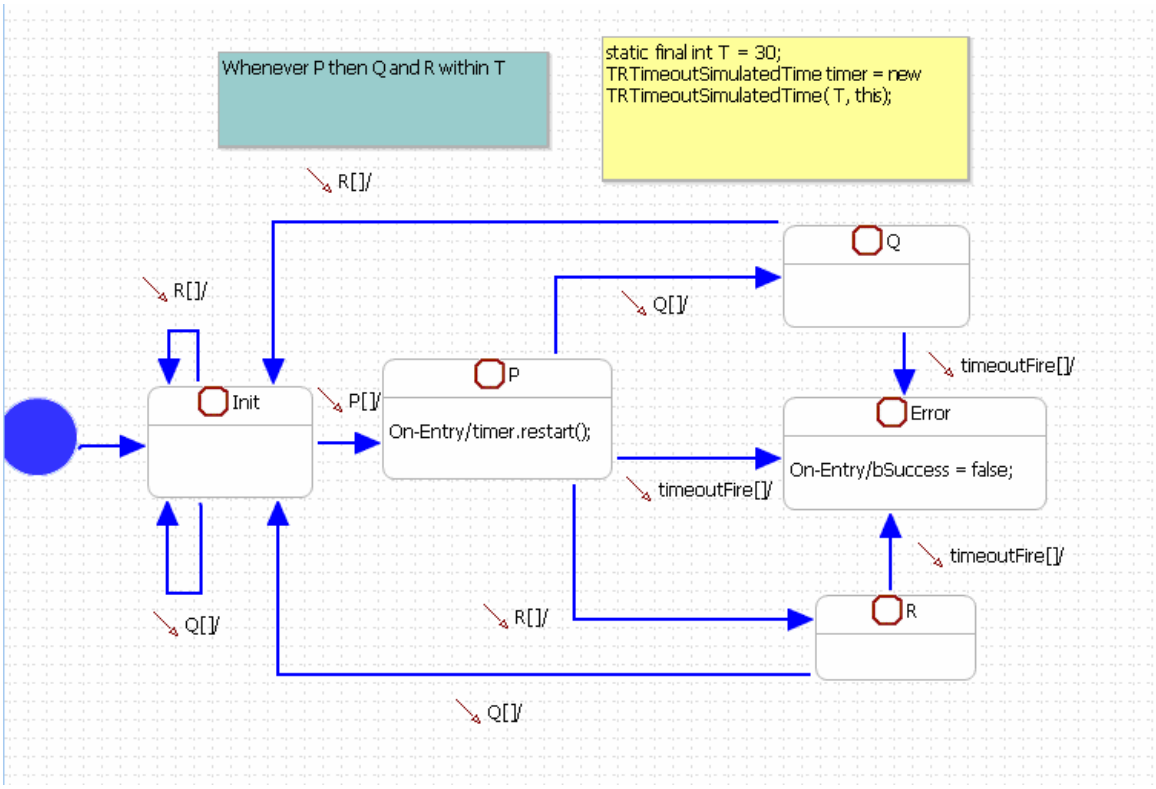


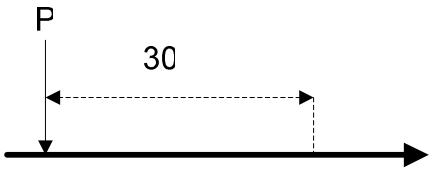
P; incrTime(5); Q; incrTime(5); Q; incrTime(5) Q; incrTime (5); P; incrTime(10); Q; incrTime(21)(timeout has occurred). Our goal in this test was to ensure that the assertion could handle overlapping time intervals and that the assertion observes more than the first P in a sequence of P’s. The test was expected to be a failure since we set N to 2. Our expectation was confirmed.



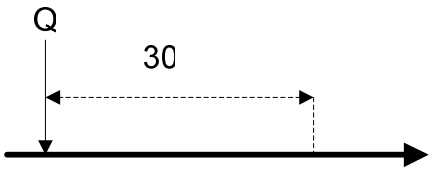
P; incrTime(5); Q; incrTime(5); Q; incrTime(5); Q; incrTime(16)(timeout has occurred); P; incrTime(5); Q; incrTime(26) (timeout has occurred). We tested for multiple intervals in this test to ensure that the assertion would observe more than a single time interval. We expected failure since we set N to 2. Our expectation was confirmed.

6. Whenever P Then Q and R Within T

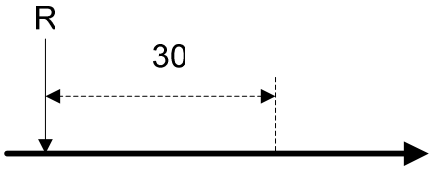




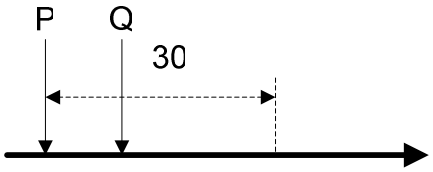
P; incrTime(31) (timeout has occurred). We expected failure. Our expectation was confirmed.



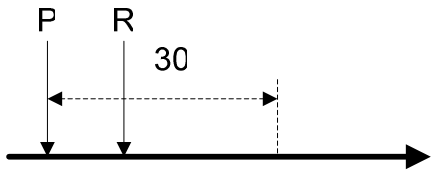
Q; incrTime(31) (timeout has occurred). We expected success. Our expectation was confirmed.



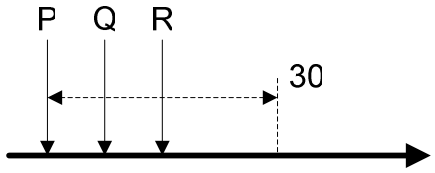
R; incrTime(31) (timeout has occurred). We expected success. Our expectation was confirmed.



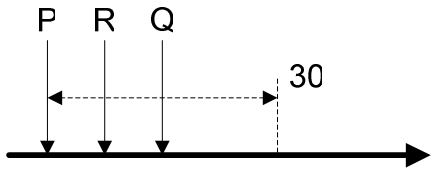
P; incrTime(5); Q; incrTime(26) (timeout has occurred). We expected failure. Our expectation was confirmed.



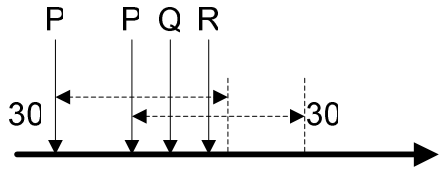
P; incrTime(5); R; incrTime(26) (timeout has occurred). We expected failure. Our expectation was confirmed.



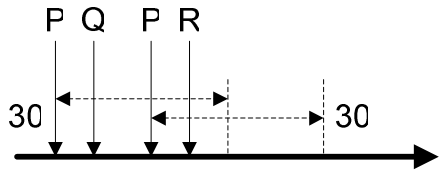
P; incrTime(5); Q; incrTime(5); R; incrTime(21)(timeout has occurred). We expected success. Our expectation was confirmed.



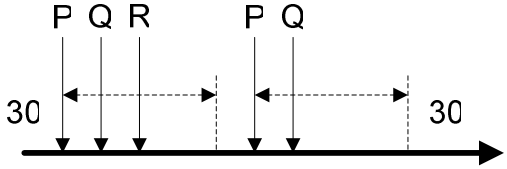
P; incrTime(5); R; incrTime(5); Q; incrTime(21)(timeout has occurred). We expected success. Our expectation was confirmed.



P; incrTime(5);P; incrTime(5); Q; incrTime (5); R; incrTime(26)(timeout has occurred). Our goal in this test and the next was to ensure that the assertion could handle overlapping time intervals and that the assertion observes more than the first P in a sequence of Ps. The test was expected to be a success. Our expectation was confirmed.



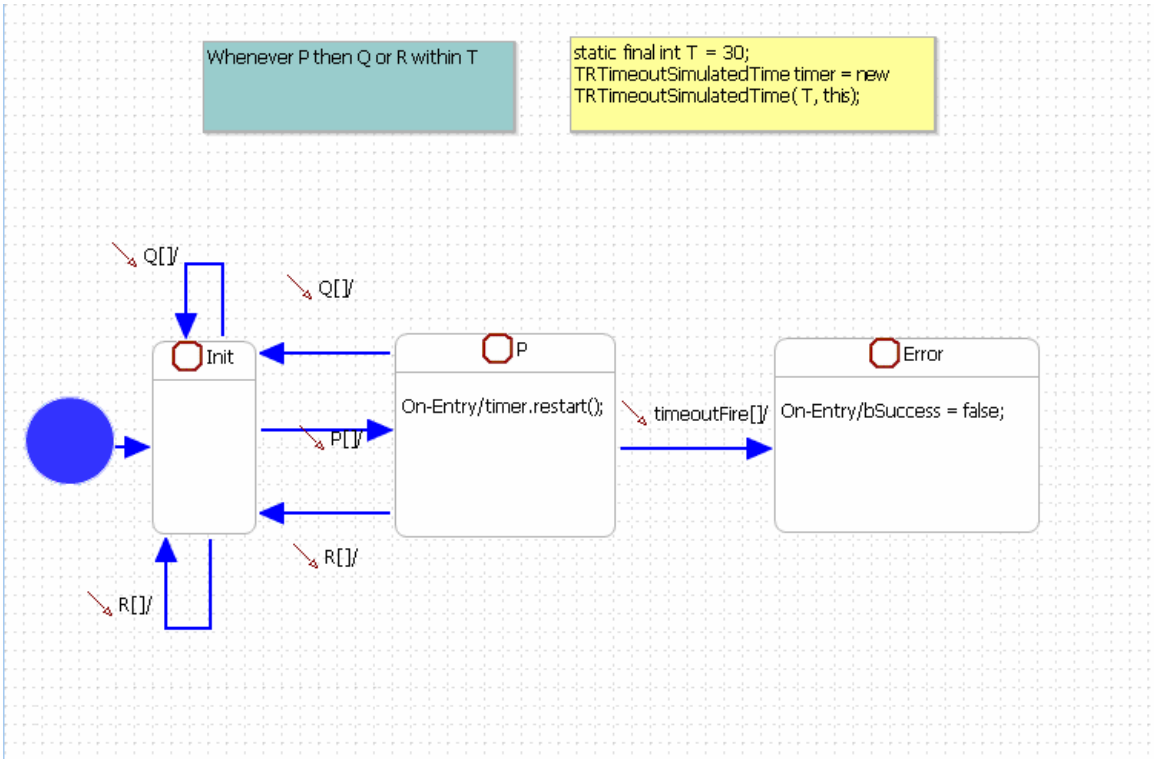
P; incrTime(5);Q; incrTime(10) P; incrTime (5); R; incrTime(26)(timeout has occurred). The test was expected to be a failure. Our expectation was confirmed.

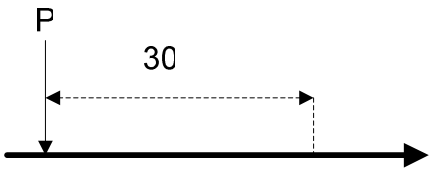


P; incrTime(5); Q; incrTime(5); R; incrTime(21)(timeout has occurred); P; incrTime(5); Q; incrTime(26) (timeout has occurred). We expected failure. Our expectation was confirmed.

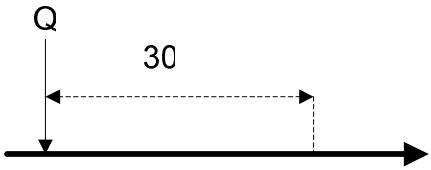
In this assertion statement we did not differentiate between Q or R coming first, our intention was to ensure that the combination of both Q and R regardless of order resulted in a successful test.

7. Whenever P Then Q or R Within T

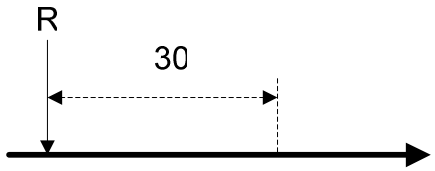




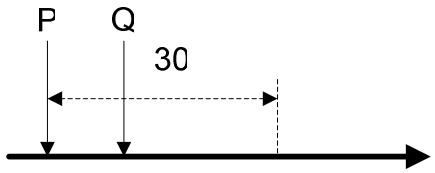
P; incrTime(31) (timeout has occurred). We expected failure. Our expectation was confirmed.



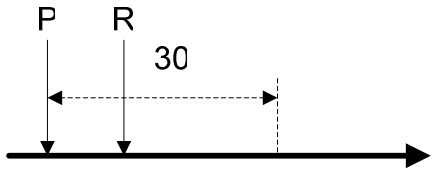
Q; incrTime(31) (timeout has occurred). We expected success. Our expectation was confirmed.



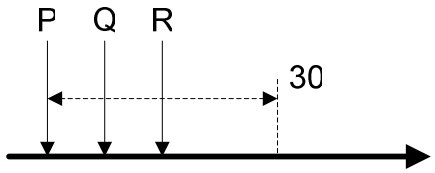
R; incrTime(31) (timeout has occurred). We expected success. Our expectation was confirmed.



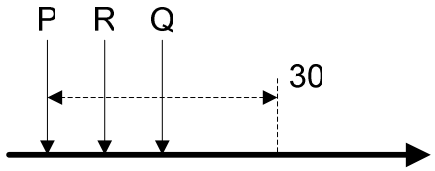
P; incrTime(5); Q; incrTime(26) (timeout has occurred). We expected success. Our expectation was confirmed.



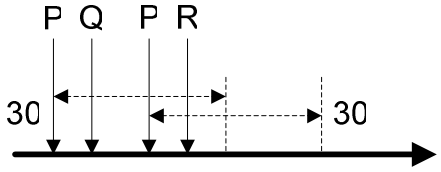
P; incrTime(5); R; incrTime(26) (timeout has occurred). We expected success. Our expectation was confirmed.



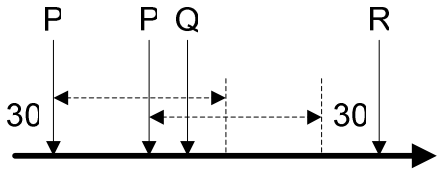
P; incrTime(5); Q; incrTime(5); R; incrTime(21)(timeout has occurred). We expected success. Our expectation was confirmed.



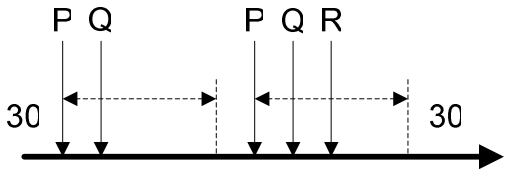
P; incrTime(5); R; incrTime(5); Q; incrTime(21)(timeout has occurred). We expected success. Our expectation was confirmed.



P; incrTime(5);Q; incrTime(15); P; incrTime(5); R; incrTime(26)(timeout has occurred). Our goal in this test and the next was to ensure that the assertion could handle overlapping time intervals and that the assertion observes more than the first P in a sequence of P's. The test was expected to be a success. Our expectation was confirmed.

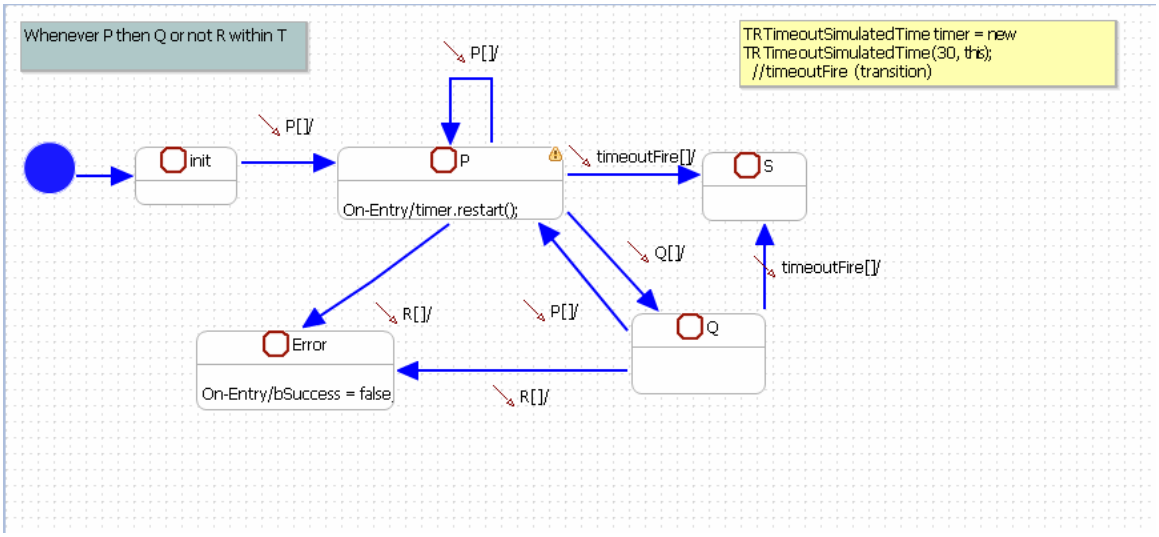


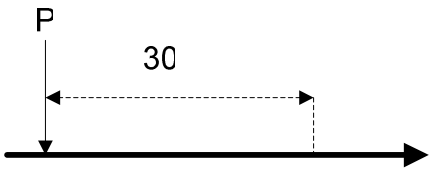
P; incrTime(5);Q; incrTime(10) P; incrTime(31) (timeout has occurred); R. The test was expected to be a success. Our expectation was confirmed.



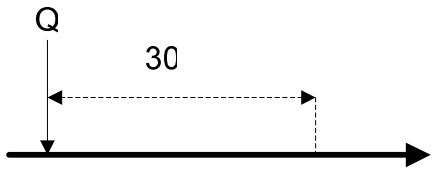
P; incrTime(5); Q; incrTime(31)(timeout has occurred); P; incrTime(5); Q; incrTime(5); R; incrTime(21) (timeout has occurred). We tested for multiple intervals in this test and the next to ensure that the assertion would observe more than a single time interval. We expected success. Our expectation was confirmed.

8. Whenever P Then Q or Rot R Within T

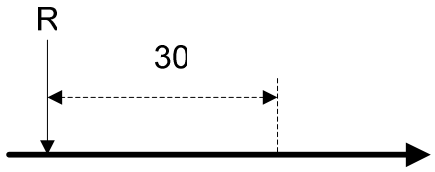




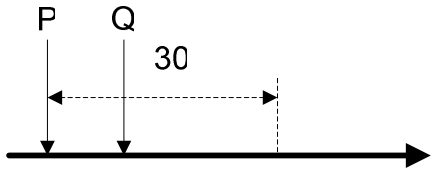
P; incrTime(31) (timeout has occurred). We expected obvious success. Our expectation was confirmed.



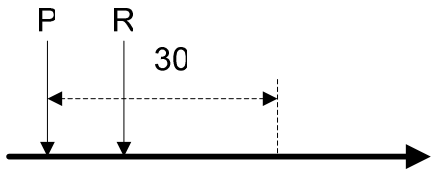
Q; incrTime(31) (timeout has occurred). We expected success. Our expectation was confirmed.



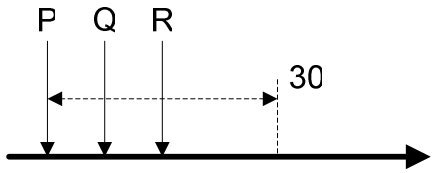
R; incrTime(31) (timeout has occurred). We expected success. Our expectation was confirmed.



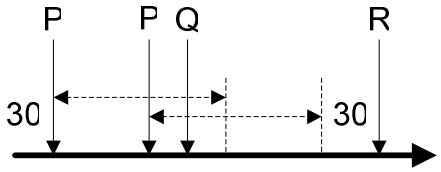
P; incrTime(5); Q; incrTime(26) (timeout has occurred). We expected success. Our expectation was confirmed.



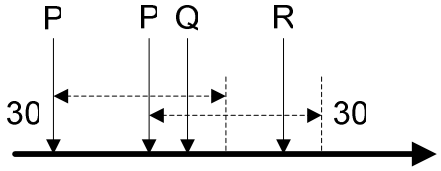
P; incrTime(5); R; incrTime(26) (timeout has occurred). We expected failure. Our expectation was confirmed.



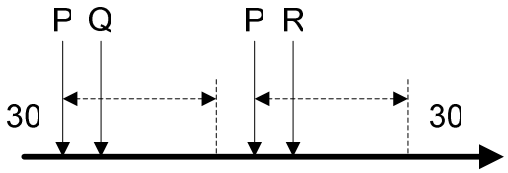
P; incrTime(5); Q; incrTime(5); R; incrTime(21)(timeout has occurred). We expected failure. Our expectation was confirmed.



P; incrTime(15);P; incrTime(5); Q; incrTime(26)(timeout has occurred); R. Our goal in this test and the next was to ensure that the assertion could handle overlapping time intervals and that the assertion observes more than the first P in a sequence of P’s. The test was expected to be a success. Our expectation was confirmed.

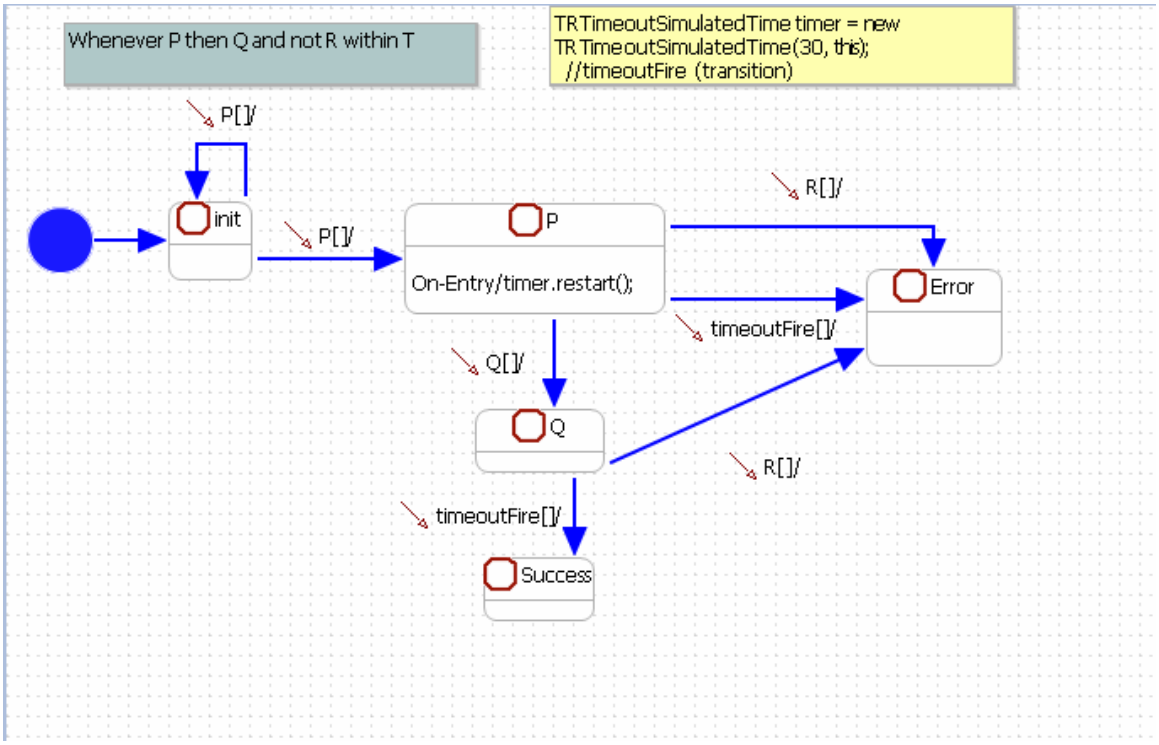


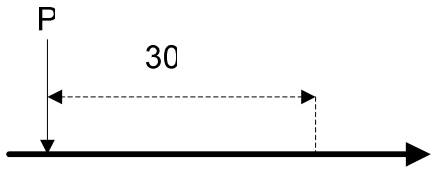
P; incrTime(5);Q; incrTime(10) P; incrTime(21); R; incrTime(10) (timeout has occurred). The test was expected to be a failure. Our expectation was confirmed.



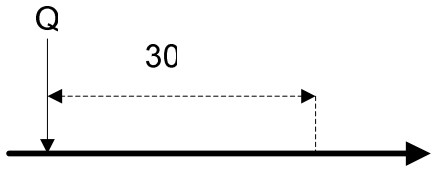
P; incrTime(5); Q; incrTime(26) (timeout has occurred); P; incrTime(5); R; incrTime(26) (timeout has occurred). We tested for multiple intervals in this test to ensure that the assertion would observe more than a single time interval. We expected failure. Our expectation was confirmed.

9. Whenever P Then Q and Not R within T

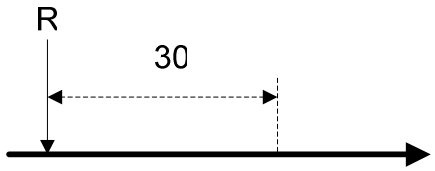




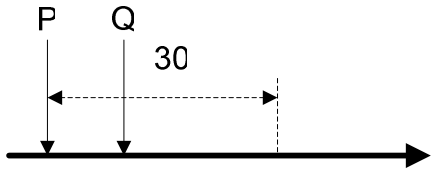
P; incrTime(31) (timeout has occurred). We expected obvious failure. Our expectation was confirmed.



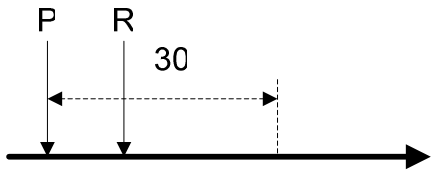
Q; incrTime(31) (timeout has occurred). We expected success. Our expectation was confirmed.



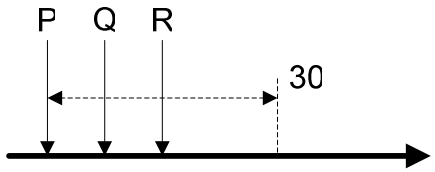
R; incrTime(31) (timeout has occurred). We expected success. Our expectation was confirmed.



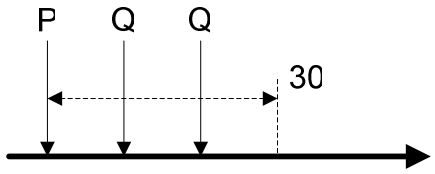
P; incrTime(5); Q; incrTime(26) (timeout has occurred). We expected success. Our expectation was confirmed.



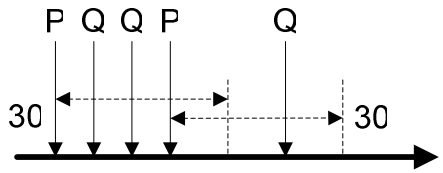
P; incrTime(5); R; incrTime(26) (timeout has occurred). We expected failure. Our expectation was confirmed.



P; incrTime(5); Q; incrTime(5); R; incrTime(21)(timeout has occurred). We expected failure. Our expectation was confirmed.

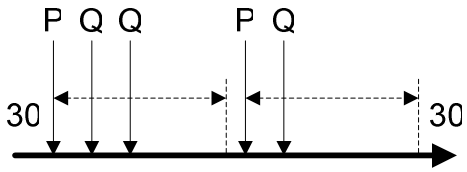


P; incrTime(5); Q; incrTime(5); Q; incrTime(21) (timeout has occurred). We expected success. Our expectation was confirmed.



P; incrTime(5); Q; incrTime(5); Q; incrTime(5); P; incrTime(20); Q; incrTime(15) (timeout has occurred). Our goal in this test was to ensure that the assertion could handle overlapping time intervals and that the assertion

observes more than the first P in a sequence of P's. The test was expected to be a success. Our expectation was confirmed.

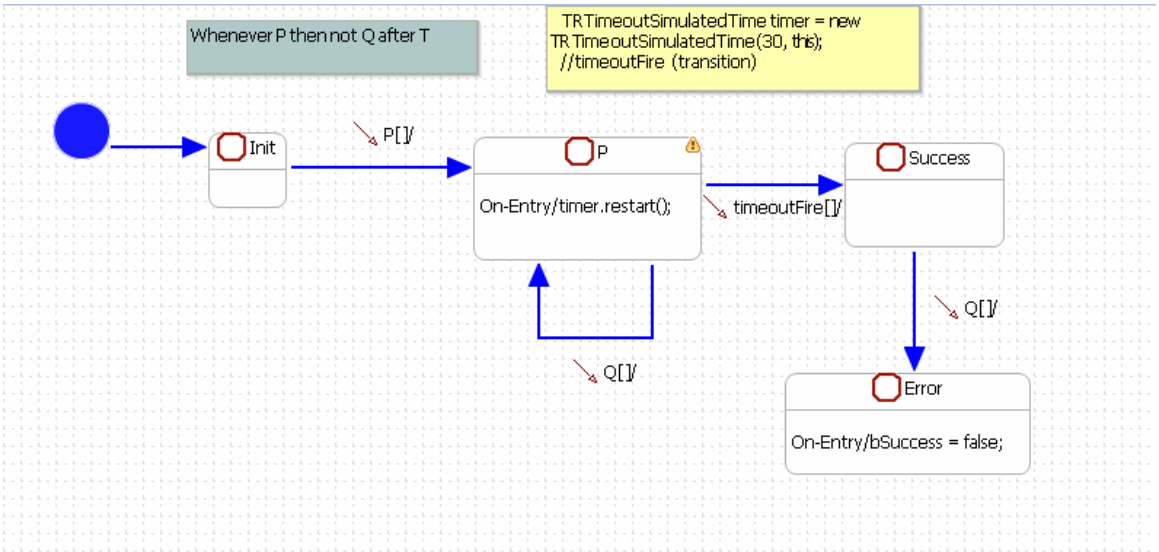


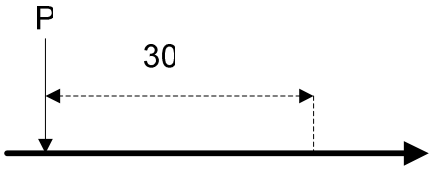
P; incrTime(5); Q; incrTime(5); Q; incrTime(21) (timeout has occurred); P; incrTime(5); Q; incrTime(26) (timeout has occurred). We tested for multiple intervals in this test to ensure that the assertion would observe more than a single time interval. We expected success. Our expectation was confirmed.

B. ADDITIONAL ASSERTION DIAGRAMS UNBOUNDED BY TIME

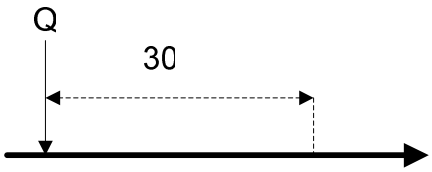
We felt that this assertion statement should be separated from the other assertion statements because the time is unbounded. It still has merit as an assertion statement but may not be as useful as the other assertions.

1. Whenever P Then Not Q After T

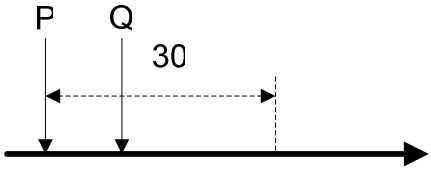




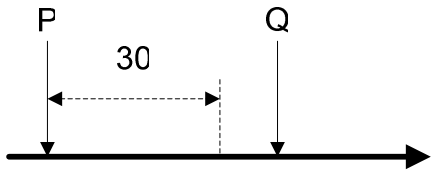
P; incrTime(31) (timeout has occurred). We expected obvious success. Our expectation was confirmed.



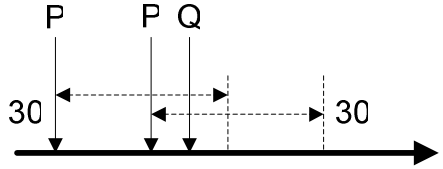
Q; incrTime(31) (timeout has occurred). We expected success. Our expectation was confirmed.



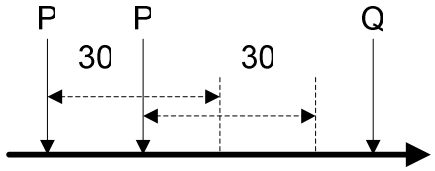
P; incrTime(5); Q; incrTime(26) (timeout has occurred). We expected success. Our expectation was confirmed.



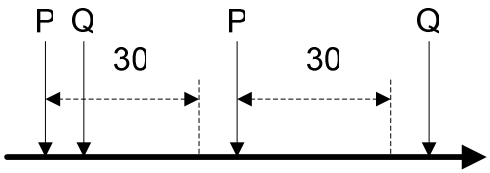
P; incrTime(31) (timeout has occurred); Q . We expected failure. Our expectation was confirmed.



P; incrTime(10); P; incrTime(10); Q; incrTime(21)(timeout has occurred). Our goal in this test was to ensure that the assertion could handle overlapping time intervals and that the assertion observes more than the first P in a sequence of Ps. We expected success. Our expectation was confirmed.



P; incrTime(5); P; incrTime(31)(timeout has occurred); Q. We expected failure. Our expectation was confirmed.



P; incrTime(5); Q; incrTime(26)(timeout has occurred); P; incrTime(31)(timeout has occurred); Q. We tested for multiple intervals in this test to ensure that the assertion would observe more than a single time interval. We expected failure. Our expectation was confirmed.

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

Appleton, B. "Patterns and software: essential concepts and terminology" *CM Crossroads* (2000), http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html (accessed September 20, 2008).

Cheng, B. and Konrad, S. "Real-time specification patterns." Proceedings of the 27th international conference on Software engineering (2005): 372-381.

Drusinsky,D. *Modeling and verification using UML statecharts - a working guide to reactive system design, runtime monitoring and execution-based model checking.* Elsevier, 2006.

D.Drusinsky, M.Shing, K.Demir, "Creation and validation of embedded assertion statecharts", *Proc. 15th IEEE International Workshop in Rapid System Prototyping*, Greece (June 14-16, 2006): 17-23.

Dwyer, M., Avrunin, G., et.al. "Patterns in property specifications for finite-state verification." Proceedings of the 21st international conference on software engineering(1999):411-420.

Harel,D. *Statecharts: A visual approach for complex systems, science of computer programming*, vol.8, no.3. 1987: 231-274.

Institute of Electrical and Electronics Engineers, Standard for Software Verification and Validation, IEEE-STD-1012, June 08, 2005.

Le Vie, D., "Writing software requirements specifications" TECHWR-L (MAR 2007) http://www.techwrl.com/techwhirl/magazine/writing/softwareequirementspecs.htm (accessed July 15, 2008).

Levenson, N. and Turner, C., "Investigation of the Therac 25 accidents." *IEEE Computer* (July 1993): 18-41.

Lewis, R. *Independent verification & validation: A life cycle engineering process for quality software.* New York: John Wiley & Sons, 1992, xxiii.

Lim, W. *Managing software reuse, a comprehensive guide to strategically reengineering the organization for reusable components.* Upper Saddle River, New Jersey: Prentice Hall PTR, 1998, 7.

Logan, R., and Nitta, C., “Verification & validation: process and levels leading to qualitative or quantitative validation statements.” *SAE Transactions* vol.113, no.5 (2004), http://bill.cacr.caltech.edu/valworkshop/upload/files/UCRLTR-200131sae04fa.pdf (accessed June 01, 2008).

Merritt, R., “Embedded experts: fix code bugs or cost lives.” *Information Week* (10 APR 2006), http://www.informationweek.com/news/management/showArticle.jhtml?articleID=185300011 (accessed May 05, 2008).

National Institute of Standards and Technology (NIST). “Software errors cost U.S. economy $59.5 billion annually.” National Institute of Standards and Technology (NIST2002-10) (2002), http://www.nist.gov/public_affairs/releases/n02-10.htm (accessed May 10, 2008).

National Aeronautics and Space Administration (NASA), "NASA IV&V facility - about IV&V." National Aeronautics and Space Administration (NASA), http://www.nasa.gov/centers/ivv/ about/index.html (accessed June 01, 2008).

NASA IV&V Facility, “NASA IV&V 2006 annual report.” NASA IV&V Facility, http://www.nasa.gov/centers/ivv/pdf/174321main_Annual_Report_06_Final.pdf (accessed June 01, 2008).

Nickolett, C., “Project due diligence: independent verification and validation.” White Paper.Comprehensive Consulting Solutions. Mar 2001: 1-6. http://www.comp-soln.com/IVV_whitepaper.pdf (accessed June 01, 2008).

Otani, T., Drusinsky, D., et.al, “Validating UML statechart based assertions libraries for improved reliability and assurance.” Proceedings of the Second International Conference on Secure System Integration and Reliability Improvement (SSIRI 2008), Yokohama, Japan (July 14-17 2008): 47-51.

Rakin, S., “Food for thought: What is software quality assurance?” *Software Quality Consulting* (Jan. 2005, Vol.2 No.1),http://www.swqual.com/newsletter/vol2/no1/vol2no1.html (accessed June 05, 2008).

Reiss, S. *A practical introduction to software design with C++.* New York: John Wiley & Sons, 1998, 397-421.

Robat, C., “Introduction to software history.” The History of Computing Project (October 17, 2006), http://www.thocp.net/software/software_reference/introduction_to_software_history.htm (accessed June 11, 2008).

The Standish Group International, "Annual Chaos Report." (2006).

Woodham, K. *System Reference Model (SRM) development and analysis guideline*, 1st draft (National Aeronautics and Space Administration (NASA), 2007).

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Dudley Knox Library
   Naval Postgraduate School
   Monterey, CA

2. Defense Technical Information Center
   Ft. Belvoir, VA

3. Professor Bret Michael
   Naval Postgraduate School
   Monterey, CA

4. Professor Man-Tak Shing
   Naval Postgraduate School
   Monterey, CA

5. Professor Doron Drusinsky
   Naval Postgraduate School
   Monterey, CA

6. Professor Tom Otani
   Naval Postgraduate School
   Monterey, CA

7. Dr. Butch Caffall
   NASA IV&V Facility
   Fairmont, WV

8. Mr. Steve Raque
   NASA IV&V Facility
   Fairmont, WV